Generación procedural de niveles en un videojuego de plataformas

Raso Vázquez, Ander Tutor: Villamañe Gironés, Mikel

25 de julio de 2021

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea

Resumen

En el mundo del desarrollo de videojuegos la generación procedural de contenido es una técnica que se usa tanto para ahorrar trabajo en la creación de niveles como para aumentar la rejugabilidad. Es común verla utilizada por estudios de desarrollo independientes para ahorrar tiempo y dinero en los diseños de niveles. En este proyecto se van a estudiar técnicas de generación procedural para crear un juego de plataformas en el que cada partida será diferente de la anterior. El jugador se moverá por tres niveles llenos de enemigos y recursos que obtener. Podrá combinar los objetos obtenidos para crear equipamiento más potente y lo usará para vencer al jefe final en el nivel 4.

Abstract

In the world of video game development, procedural content generation is a technique used to save work effort on level creation and to increase replayability. It is common to see it used by independent development studios in order to save time and money on level designs. In this project, procedural generation techniques will be studied to create a platform game where each playthrough will be different from the previous one. The player will move through three levels full of enemies and resources to obtain. He will be able to combine the items obtained to create

more powerful equipment and use it to defeat the final boss at level 4.

Laburpena

Bideo-jokoen garapenaren munduan, edukia prozedurala sortzea lana aurrezteko eta berriro jokatzea dibertigarriago izateko erabiltzen den teknika bat da. Ohikoa da garapen-estudio independenteek erabiltzea, mundu diseinuetan denbora eta dirua aurrezteko. Proiektu honetan, eduki prozedurala sortzeko teknikak aztertuko dira, plataforma-joko bat sortzeko, non partida bakoitza aurrekoaren desberdina izango den. Aurkariz eta baliabidez betetako hiru mundutan mugituko da jokalaria. Lortutako objektuak konbinatu ahal izango ditu ekipamendu indartsuagoa sortzeko, eta 4. munduko azken burua garaitzeko erabiliko du.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

La idea principal es hacer un videojuego en el que cada partida proporcione una experiencia distinta a la anterior. Para ello, los niveles se crearán de forma aleatoria respetando una serie de normas, nunca pudiendo jugar a dos niveles iguales. Controlaremos a un personaje que utilizará materiales del entorno y objetos obtenidos al destruir enemigos que utilizará para mejorar su equipo con el objetivo de sobrevivir hasta derrotar al jefe final.

1.1. Origen del proyecto

Uno de mis pasatiempos favoritos es apuntarme a competiciones de programación. Hay unos eventos de desarrollar videojuegos que se llaman *Game Jams* y consisten en lo siguiente:

- Los organizadores el mismo día de la competición informan del tema en el que se tendrá que basar un videojuego. Normalmente ese tema suele ser una palabra o frase corta. Por ejemplo: "Tú eres tu enemigo", usando esta temática como punto de partida, cada persona o grupo tendría que interpretar esa frase como lo vea preciso para hacer un videojuego.
- Desde ese mismo instante los participantes disponen de un tiempo determinado para terminar el juego.
- El tiempo límite varía en cada competición, puede ser meses, días o incluso tan poco como una hora.

- Algunos Game Jams también imponen otras restricciones especiales, como el GBJAM en el que la pantalla tiene que tener las dimensiones de una consola Game Boy y se debe utilizar una paleta de 4 colores. [2]

La idea para este proyecto surgió cuando estaba aprendiendo a crear un *tileset* para un Game Jam. Un tileset es un recurso compuesto de unos cuadraditos llamados *tiles* y su función es la de crear mapas dibujando esos cuadrados (o tiles) en él. Lo interesante de los tileset es que sus tiles pueden detectar los tiles que tienen a su alrededor y así pintar una forma en concreto.

Cuando terminé de aprender a crear el tileset pensé que podría ser divertido, para un proyecto posterior, averiguar cómo usar esta funcionalidad para que se creasen niveles automáticamente por código. De esta forma podría hacer un videojuego en el que no tuviese que crear yo manualmente los niveles. Además, añadiendo un poco de aleatoriedad a la creación también podría hacer que cada vez que jugase obtuviese una experiencia diferente.

1.2. Motivaciones para la elección del proyecto

Un proyecto de este tipo es una de las mejores formas de demostrar los conocimientos que se han ido adquiriendo durante la carrera, ya que requiere una integración de prácticamente todas las áreas de conocimiento que hemos estudiado. Por poner algún ejemplo, veremos como se aplica el álgebra vectorial para el movimiento de los elementos del videojuego, se utilizarán buenas prácticas en diseño de software que permitirán crear soluciones tan fáciles de mantener como extensibles y se estudiará qué métodos usar para el almacenamiento de información.

1.3. Situación del trabajo

Este proyecto no se inicia desde cero. Para proponer la idea se desarrolló un pequeño prototipo en el que se mostraba la creación de un nivel muy básico en el que se podía mover a un personaje por él talando árboles. En este trabajo se va a partir de esa base para desarrollar el resto de funcionalidades.

Capítulo 2

Planteamiento inicial

2.1. Objetivos

El objetivo principal de este proyecto es crear un videojuego en el que cada partida sea diferente de la anterior. Para lograrlo, los niveles se generarán proceduralmente. Esto quiere decir que en cada partida nueva la estructura del nivel, la localización de los recursos y los enemigos será aleatoria, pero seguirá un conjunto de normas básicas para tener cierto orden.

Otro objetivo es adquirir nuevos conocimientos en el mundo del desarrollo de los videojuegos. Ya que este videojuego es el proyecto más ambicioso en el que me he embarcado hasta ahora, espero que resolver los problemas que vayan apareciendo me permita aprender nuevas técnicas que poder emplear en futuros proyectos.

Por último, quiero demostrar que se puede crear un juego utilizando únicamente software de código abierto. Es decir, software cuyo código está disponible a la vista de todo el que quiera verlo.

2.2. Alcance

Ahora que se han definido los objetivos, se necesitan definir las tareas que serán necesarias realizar para cumplirlos. Como es habitual en los ciclos de desarrollo de videojuegos, se ha decidido realizar el diseño primero y dividir la parte de implementación en pequeños prototipos (PT). Primero se diseñarán las funciones, después se dibujarán los recursos y por último se realizará la implementación de los prototipos. Estos prototipos pasarán por una fase de pruebas y por último se realizará la corrección de los errores encontrados.

De esta manera, cada prototipo acercará más al desarrollo a la visión final que se ha conformado del mismo. Además, realizar una fase de pruebas por cada prototipo permite detectar errores pronto que podrán ser subsanados antes de pasar a implementar otras funciones.

Recordemos que en el estado inicial de este TFG se dispone ya de un personaje capaz de moverse. Los siguientes prototipos que se han definido añadirán funciones a lo que ya se tiene:

- **PT1. Creación de niveles**: consiste en añadir la lógica de generación procedural del nivel. Esto quiere decir que al finalizar este prototipo, al iniciar la partida el personaje aparecerá en un nivel de plataformas que debido a la aleatoriedad introducida, será diferente cada vez.
- **PT2. Elementos interactuables**: consiste en añadir a la generación de nivel elementos con los que el jugador podrá interactuar. Por ejemplo árboles que se podrán cortar, plantas que recolectar y minerales que minar.
- **PT3. Implementación de interfaces**: consiste en añadir la interfaz que muestre la vida del jugador, el inventario de los objetos en su posesión, las combinaciones posibles entre sus objetos [1] y los menús necesarios para empezar a jugar, pausar y terminar el juego.
- **PT4. Objetos utilizables**: ahora que ya se dispone de una interfaz para visualizar objetos, se implementarán los objetos que el jugador podrá utilizar. Esto incluye herramientas para talar árboles, minar minerales o dañar enemigos, poción para restaurar salud y armaduras.

[1] Un ejemplo de combinación de objetos podría ser combinar una hierba con miel para producir una poción con la que el jugador podrá restaurar parte de su salud

- **PT5. Creación de enemigos**: última fase del proyecto. Se añadirán enemigos que atacarán al personaje y que al ser debilitados soltarán objetos que el jugador podrá utilizar.

En resumen, al finalizar el proyecto se tendrá un videojuego en el que se controlará a un personaje que el jugador moverá por niveles únicos. Será capaz de utilizar los recursos del entorno para construir objetos útiles y esto le permitirá derrotar a los enemigos con más facilidad.

2.2.1. Diagrama de la estructura de la descomposición del trabajo

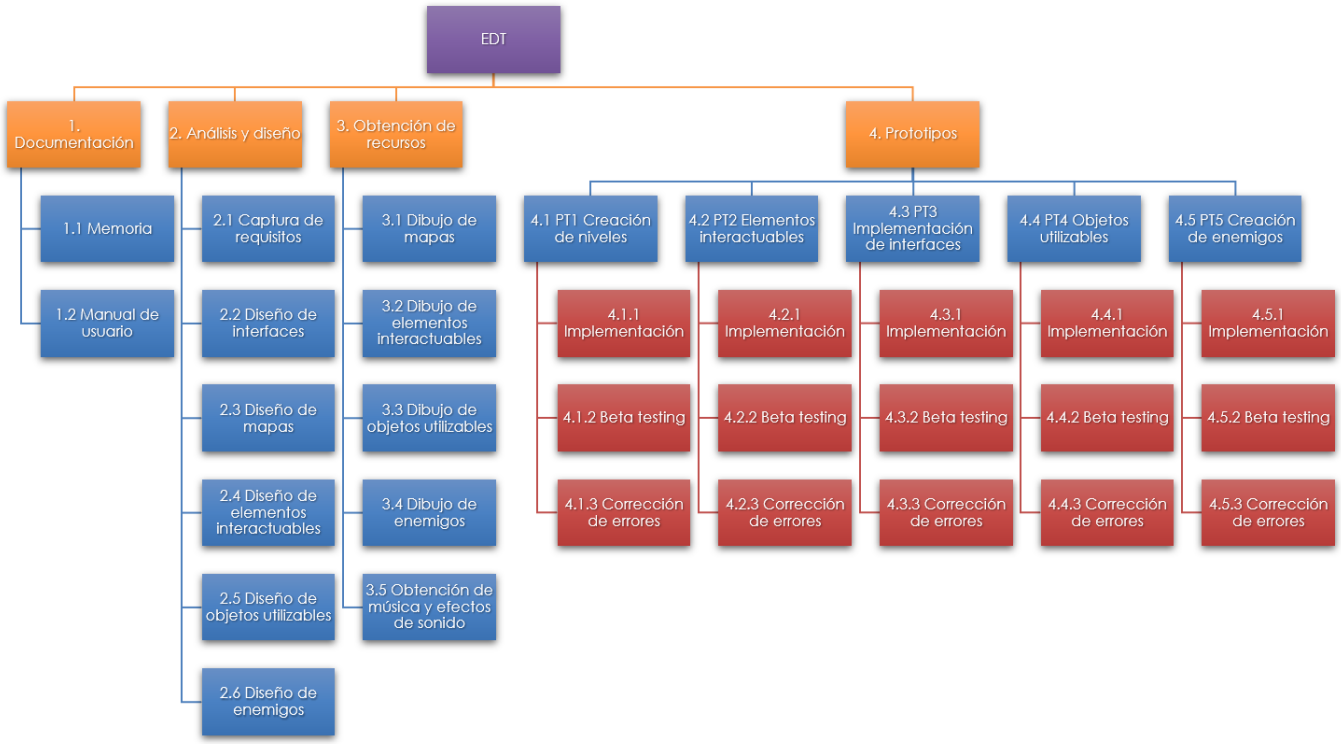

Figura 2.1: EDT

2.2.2. Descripción de las tareas

Documentación

1.1 Memoria
Paquete de trabajo: 1. Documentación **Responsable:** Ander Raso Vázquez **Duración:** 50 horas
Descripción Se plasmará toda la información del proyecto en una memoria.
Entradas N/A
Salidas/Entregables La memoria de este TFG en formato PDF.
Recursos necesarios Un PC con acceso a Internet para entrar a la página *Overleaf*, ya que se utilizará para formatear el documento mediante *Latex*
Precedencias N/A

Tabla 2.1: 1.1 Memoria

1.1 Manual de usuario
Paquete de trabajo: 1. Documentación **Responsable:** Ander Raso Vázquez **Duración:** 10 horas
Descripción Para que el jugador sepa qué es lo que puede hacer en el juego, se escribirá un manual de usuario que le indique los controles con los que mover al personaje y las combinaciones que puede realizar con los objetos.
Entradas N/A
Salidas/Entregables Un PDF con la información de cómo se juega.
Recursos necesarios Un PC con acceso a Internet para entrar a la página *Overleaf*, ya que se utilizará para formatear el documento mediante *Latex*
Precedencias 4.5.3

Tabla 2.2: 1.1 Manual de usuario

Análisis y diseño

2.1 Captura de requisitos
Paquete de trabajo: 2. Análisis y diseño **Responsable:** Ander Raso Vázquez **Duración:** 3 horas
Descripción Se crearán diagramas de casos de uso para identificar la jerarquía de actores y los requisitos que se deberán implementar.
Entradas N/A
Salidas/Entregables Diagramas de casos de usos.
Recursos necesarios Un PC con acceso a Internet a *PlantUML*, la página con la que se crearán los gráficos.
Precedencias N/A

Tabla 2.3: 2.1 Captura de requisitos

2.2 Diseño de interfaces
Paquete de trabajo: 2. Análisis y diseño **Responsable:** Ander Raso Vázquez **Duración:** 3 horas
Descripción Diseño de las interfaces que necesitará el jugador para empezar partida, pausar el juego, salir del juego, combinar objetos y ver su inventario.
Entradas N/A
Salidas/Entregables Diagramas de clase y de secuencia.
Recursos necesarios Un PC con acceso a Internet a *PlantUML*, la página con la que se crearán los gráficos.
Precedencias N/A

Tabla 2.4: 2.2 Diseño de interfaces

2.3 Diseño de mapas

Paquete de trabajo: 2. Análisis y diseño
Responsable: Ander Raso Vázquez
Duración: 3 horas

Descripción
Lógica para generar niveles de forma procedural. Se deberá investigar qué método de generación aleatorio es el más idóneo para este proyecto y crear los diagramas necesarios para implementarlo.

Entradas
N/A

Salidas/Entregables
Diagramas de clase y de secuencia.

Recursos necesarios
Un PC con acceso a Internet a *PlantUML*, la página con la que se crearán los gráficos.

Precedencias
N/A

Tabla 2.5: 2.3 Diseño de mapas

2.4 Diseño de elementos interactuables

Paquete de trabajo: 2. Análisis y diseño
Responsable: Ander Raso Vázquez
Duración: 10 horas

Descripción
Incluye el diseño de los elementos con los que el jugador puede interactuar. Árboles, hierbas en el suelo, setas en el suelo, panales de miel y minerales (hierro y oro).

Entradas
N/A

Salidas/Entregables
Diagramas de clase y de secuencia.

Recursos necesarios
Un PC con acceso a Internet a *PlantUML*, la página con la que se crearán los gráficos.

Precedencias
N/A

Tabla 2.6: 2.4 Diseño de elementos interactuables

2.5 Diseño de objetos utilizables

Paquete de trabajo: 2. Análisis y diseño
Responsable: Ander Raso Vázquez
Duración: 10 horas

Descripción

Incluye el diseño de los objetos que podrá utilizar el jugador. Armaduras (hierro, oro), espadas (hierro, oro), picos (madera, hierro), madera, hierbas cortadas, setas recogidas, pepitas de minerales (hierro y oro), trozo de miel, poción que recupera salud.

Entradas

Diagramas de clase de los objetos interactuables para decidir qué objetos utilizables le darán al personaje.

Salidas/Entregables

Diagramas de clase y de secuencia.

Recursos necesarios

Un PC con acceso a Internet a *PlantUML*, la página con la que se crearán los gráficos.

Precedencias

2.4

Tabla 2.7: 2.5 Diseño de objetos utilizables

2.6 Diseño de enemigos

Paquete de trabajo: 2. Análisis y diseño
Responsable: Ander Raso Vázquez
Duración: 15 horas

Descripción

Diseño de los enemigos con los que el jugador se debe enfrentar:

- **Esqueleto:** se acerca al jugador cuando lo detecta.
- **Polilla:** se acerca al jugador volando, movimiento en X e Y.
- **2 Babosas:** saltitos aleatorios o hacia el jugador.
- **Goblin:** ataca al jugador si se acerca lo suficiente.
- **Goblin mago:** atacará desde lejos lanzando una bola de magia.
- **Ogro:** jefe final. Se avalanza al jugador periódicamente.

Entradas

N/A

Salidas/Entregables

Diagramas de clase y de secuencia.

Recursos necesarios

Un PC con acceso a Internet a *PlantUML*.

Precedencias

N/A

Tabla 2.8: 2.6 Diseño de enemigos

Obtención de recursos

3.1 Dibujo del mapa

Paquete de trabajo: 3. Obtención de recursos
Responsable: Ander Raso Vázquez
Duración: 6 horas

Descripción
Se dibujarán los tilesets (el conjunto de cuadraditos que permiten dibujar mapas) de los 3 niveles. El bosque verde, el bosque de champiñones y el castillo.

Entradas
N/A

Salidas/Entregables
Salidas

Recursos necesarios
Un PC con el programa Aseprite.

Precedencias
N/A

Tabla 2.9: 3.1 Dibujo del mapa

3.2 Dibujo de elementos interactuables

Paquete de trabajo: 3. Obtención de recursos
Responsable: Ander Raso Vázquez
Duración: 5 horas

Descripción
Se dibujarán los elementos descritos en 2.4.

Entradas
Diagramas de clase para identificar lo que hay que dibujar.

Salidas/Entregables
Dibujos en formato *png*.

Recursos necesarios
Un PC con el programa de dibujo *Aseprite*.

Precedencias
2.4

Tabla 2.10: 3.2 Dibujo de elementos interactuables

3.3 Dibujo de objetos utilizables
Paquete de trabajo: 3. Obtención de recursos **Responsable:** Ander Raso Vázquez **Duración:** 5 horas
Descripción Se dibujarán los elementos descritos en 2.5.
Entradas Diagramas de clase para identificar lo que hay que dibujar.
Salidas/Entregables Dibujos en formato *png*.
Recursos necesarios Un PC con el programa de dibujo *Aseprite*.
Precedencias 2.5

Tabla 2.11: 3.3 Dibujo de objetos utilizables

3.4 Dibujos de enemigos
Paquete de trabajo: 3. Obtención de recursos **Responsable:** Ander Raso Vázquez **Duración:** 9 horas
Descripción Se dibujarán los elementos descritos en 2.6.
Entradas Diagramas de clase para identificar lo que hay que dibujar.
Salidas/Entregables Dibujos en formato *png*.
Recursos necesarios Un PC con el programa de dibujo *Aseprite*.
Precedencias 2.6

Tabla 2.12: 3.4 Dibujos de enemigos

3.5 Obtención de música y objetos de sonido
Paquete de trabajo: 3. Obtención de recursos **Responsable:** Ander Raso Vázquez **Duración:** 3 horas
Descripción Se buscará en Internet efectos sonoros y música de libre uso.
Entradas Lista de diagramas de clase para identificar los posibles efectos sonoros.
Salidas/Entregables Una música de fondo por nivel y efectos de sonido.
Recursos necesarios Un PC con acceso a Internet.
Precedencias 2.2, 2.3, 2.4, 2.5, 2.6

Tabla 2.13: 3.5 Obtención de música y objetos de sonido

Prototipos

4.1.1 Implementación de la creación del nivel
Paquete de trabajo: 4. Prototipos **Responsable:** Ander Raso Vázquez **Duración:** 10 horas
Descripción Se utilizarán los diagramas para realizar la implementación procedural del nivel.
Entradas Diagramas de clase, de secuencia y los tilesets de los mapas.
Salidas/Entregables Código que implementa los diagramas y una lista de las pruebas a realizar.
Recursos necesarios Un PC con el motor de juego *Godot Engine* instalado.
Precedencias 2.3, 3.1

Tabla 2.14: 4.1.1 Implementación de la creación del nivel

4.1.2 Beta testing de la creación del nivel
Paquete de trabajo: 4. Prototipos **Responsable:** Ander Raso Vázquez **Duración:** 3 horas
Descripción Se comprobará si se crean los niveles como se ha definido.
Entradas Juego a probar y una lista de los elementos a probar.
Salidas/Entregables Informe de errores.
Recursos necesarios Un PC con el juego instalado.
Precedencias 4.1.1

Tabla 2.15: 4.1.2 Beta testing de la creación del nivel

4.1.3 Corrección de errores en la creación del nivel
Paquete de trabajo: 4. Prototipos **Responsable:** Ander Raso Vázquez **Duración:** 3 horas
Descripción Corrección de los errores obtenidos en el informe.
Entradas Informe de errores.
Salidas/Entregables Código arreglado.
Recursos necesarios Un PC con el motor de juego *Godot Engine* instalado.
Precedencias 4.1.2

Tabla 2.16: 4.1.3 Corrección de errores en la creación del nivel

4.2.1 Implementación de elementos interactuables
Paquete de trabajo: 4. Prototipos **Responsable:** Ander Raso Vázquez **Duración:** 20 horas
Descripción Se utilizarán los diagramas para realizar la implementación de los elementos interactuables.
Entradas Diagramas de clase, de secuencia y los dibujos de los elementos interactuables.
Salidas/Entregables Código que implementa los diagramas y una lista de las pruebas a realizar.
Recursos necesarios Un PC con el motor de juego *Godot Engine* instalado.
Precedencias 2.4, 3.2

Tabla 2.17: 4.2.1 Implementación de elementos interactuables

4.2.2 Beta testing de elementos interactuables
Paquete de trabajo: 4. Prototipos **Responsable:** Ander Raso Vázquez **Duración:** 3 horas
Descripción Se comprobará si la interacción con los elementos del nivel se realiza de la forma correcta.
Entradas Juego a probar y una lista de los elementos a probar.
Salidas/Entregables Informe de errores.
Recursos necesarios Un PC con el juego instalado.
Precedencias 4.2.1

Tabla 2.18: 4.2.2 Beta testing de elementos interactuables

4.2.3 Corrección de errores en elementos interactuables
Paquete de trabajo: 4. Prototipos **Responsable:** Ander Raso Vázquez **Duración:** 4 horas
Descripción Corrección de los errores obtenidos en el informe.
Entradas Informe de errores.
Salidas/Entregables Código arreglado.
Recursos necesarios Un PC con el motor de juego *Godot Engine* instalado.
Precedencias 4.2.2

Tabla 2.19: 4.2.3 Corrección de errores en elementos interactuables

4.3.1 Implementación de las interfaces

Paquete de trabajo: 4. Prototipos
Responsable: Ander Raso Vázquez
Duración: 25 horas

Descripción

Se utilizarán los diagramas para realizar la implementación de las interfaces.

Entradas

Diagramas de clase y de secuencia.

Salidas/Entregables

Código que implementa los diagramas y una lista de las pruebas a realizar.

Recursos necesarios

Un PC con el motor de juego *Godot Engine* instalado.

Precedencias

2.2

Tabla 2.20: 4.3.1 Implementación de las interfaces

4.3.2 Beta testing de las interfaces
Paquete de trabajo: 4. Prototipos **Responsable:** Ander Raso Vázquez **Duración:** 3 horas
Descripción Se comprobará si las interfaces funcionan de la forma definida.
Entradas Juego a probar y una lista de los elementos a probar.
Salidas/Entregables Informe de errores.
Recursos necesarios Un PC con el juego instalado.
Precedencias 4.3.1

Tabla 2.21: 4.3.2 Beta testing de las interfaces

4.3.3 Corrección de las interfaces
Paquete de trabajo: 4. Prototipos **Responsable:** Ander Raso Vázquez **Duración:** 5 horas
Descripción Corrección de los errores obtenidos en el informe.
Entradas Informe de errores.
Salidas/Entregables Código arreglado.
Recursos necesarios Un PC con el motor de juego *Godot Engine* instalado.
Precedencias 4.3.2

Tabla 2.22: 4.3.3 Corrección de las interfaces

4.4.1 Implementación de objetos utilizables

Paquete de trabajo: 4. Prototipos
Responsable: Ander Raso Vázquez
Duración: 20 horas

Descripción
Se utilizarán los diagramas para realizar la implementación de los objetos utilizables.

Entradas
Diagramas de clase, de secuencia y dibujos de los objetos utilizables.

Salidas/Entregables
Código que implementa los diagramas y una lista de las pruebas a realizar.

Recursos necesarios
Un PC con el motor de juego *Godot Engine* instalado.

Precedencias
2.5, 3.3

Tabla 2.23: 4.4.1 Implementación de objetos utilizables

4.4.2 Beta testing de los objetos utilizables
Paquete de trabajo: 4. Prototipos **Responsable:** Ander Raso Vázquez **Duración:** 3 horas
Descripción Se comprobará si los objetos utilizables funcionan de la forma definida.
Entradas Juego a probar y una lista de los elementos a probar.
Salidas/Entregables Informe de errores.
Recursos necesarios Un PC con el juego instalado.
Precedencias 4.4.1

Tabla 2.24: 4.4.2 Beta testing de los objetos utilizables

4.4.3 Corrección de errores de los objetos utilizables
Paquete de trabajo: 4. Prototipos **Responsable:** Ander Raso Vázquez **Duración:** 4 horas
Descripción Corrección de los errores obtenidos en el informe.
Entradas Informe de errores.
Salidas/Entregables Código arreglado.
Recursos necesarios Un PC con el motor de juego *Godot Engine* instalado.
Precedencias 4.4.2

Tabla 2.25: 4.4.3 Corrección de errores de los objetos utilizables

4.5.1 Implementación de enemigos

Paquete de trabajo: 4. Prototipos
Responsable: Ander Raso Vázquez
Duración: 35 horas

Descripción

Se utilizarán los diagramas para realizar la implementación de los enemigos.

Entradas

Diagramas de clase, de secuencia y dibujos de los enemigos.

Salidas/Entregables

Código que implementa los diagramas y una lista de las pruebas a realizar.

Recursos necesarios

Un PC con el motor de juego *Godot Engine* instalado.

Precedencias

2.6, 3.4

Tabla 2.26: 4.5.1 Implementación de enemigos

4.5.2 Beta testing de enemigos
Paquete de trabajo: 4. Prototipos **Responsable:** Ander Raso Vázquez **Duración:** 3 horas
Descripción Se comprobará si los enemigos se comportan de la forma definida.
Entradas Juego a probar y una lista de los elementos a probar.
Salidas/Entregables Informe de errores.
Recursos necesarios Un PC con el juego instalado.
Precedencias 4.5.1

Tabla 2.27: 4.5.2 Beta testing de enemigos

4.5.3 Corrección de errores en enemigos
Paquete de trabajo: 4. Prototipos **Responsable:** Ander Raso Vázquez **Duración:** 7 horas
Descripción Corrección de los errores obtenidos en el informe.
Entradas Informe de errores.
Salidas/Entregables Código arreglado.
Recursos necesarios Un PC con el motor de juego *Godot Engine* instalado.
Precedencias 4.5.2

Tabla 2.28: 4.5.3 Corrección de errores en enemigos

2.3. Planificación temporal

Para poder realizar la planificación temporal del proyecto se necesitaba definir las tareas que se van a realizar para completarlo, sus tareas predecesoras y la estimación del tiempo que se va a tardar en finalizarlas.

Cuando se tienen tareas que no tienen predecesoras y se dispone de más de un trabajador, se pueden realizar esas tareas en paralelo. En este caso, como yo soy el único participante, se ha tenido que determinar una tarea predecesora en el diagrama de Gantt para las tareas que no dependían de la finalización de otras. Un ejemplo de ello es la tarea *3.1 Dibujo del mapa*, que no tiene predecesoras y se ha decidido que empezará al terminar la tarea *2.6 Diseño de enemigos*. Sin embargo, una tarea que se va a realizar en paralelo es la documentación de la memoria. Ya que según se van realizando los diagramas y se va desarrollando el proyecto se irá apuntando todo.

A continuación se muestra el diagrama de Gantt, detallando las horas por tarea y las fechas de inicio/finalización:

- **Total de horas estimadas:** 278 horas.
- **Jornada de trabajo:** Debido a que tengo que trabajar e ir a la universidad, se han estimado 2 horas diarias de trabajo para intentar limitar el riesgo *RP3 Poco tiempo disponible*, que se explicará en detalle en el apartado *2.5 Gestión de riesgos*.
- **Días a trabajar:** De lunes a viernes.
- **Fecha de inicio:** 1 de octubre de 2020.
- **Fecha de finalización:** 28 de abril de 2021.

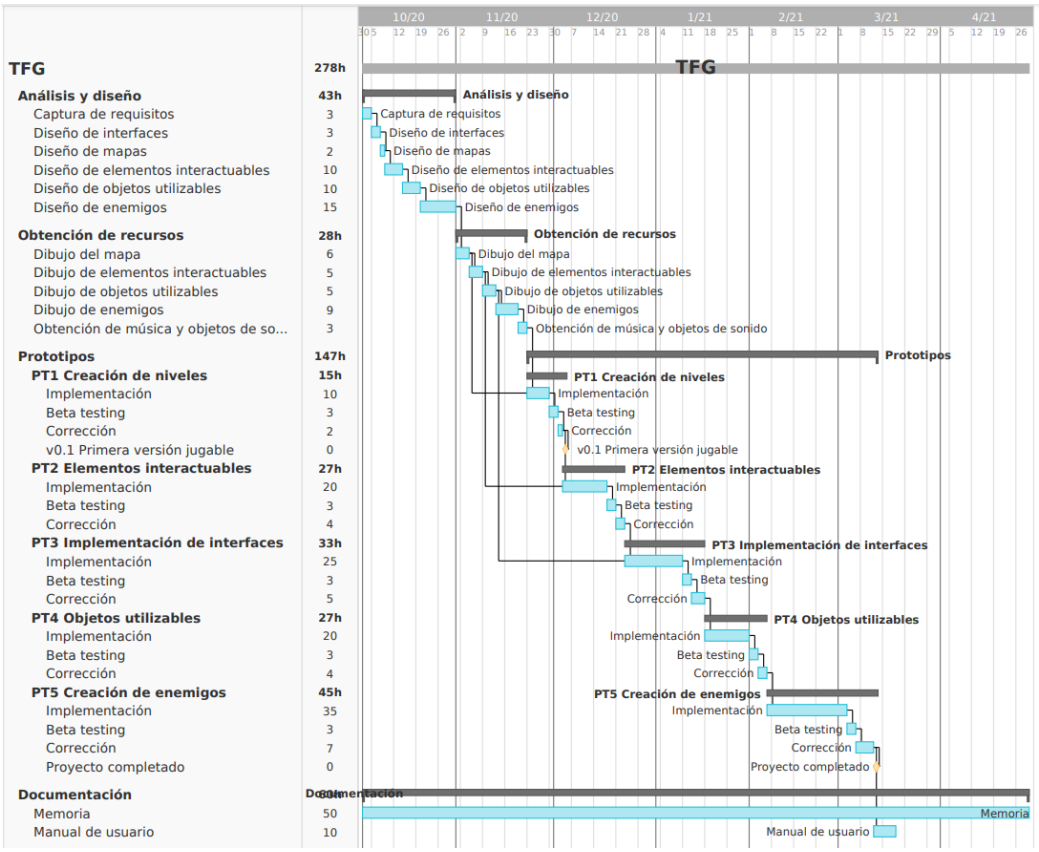

Figura 2.2: Diagrama de Gantt

2.4. Gestión de riesgos

En un mundo ideal, los desarrollos se diseñarían, se implementarían y se probarían tranquilamente. Sin embargo, en el ciclo de vida de un proyecto, con toda probabilidad surgirán contratiempos inesperados que invaliden las estimaciones que se han realizado o que incluso puedan destruir definitivamente el proyecto.

Es por eso que es necesario llevar una correcta gestión de los riesgos para poder ser capaces de identificarlos, actuar si se producen y que de esa manera no se materialicen en consecuencias no deseadas.

A continuación se muestran los tipos de riesgos que han sido identificados:

- **RP**: Riesgos del personal.
- **RH**: Riesgos del hardware, el equipo con el que se trabaja.
- **RD**: Riesgos del desarrollo.

RP1. Contagiarse de COVID-19	
Descripción	Viviendo en época de pandemia mundial de COVID-19 es necesario tomar las medidas sanitarias adecuadas de prevención, además siendo una persona asmática es posible que los efectos sean mayores.
Prevención	Se seguirán atentamente las indicaciones del personal sanitario y se verificará la información que se obtenga de agentes externos para no caer en las típicas mentiras que circulan por Internet. Ejemplos de medidas a tomar: ▪ Llevar mascarilla correctamente colocada. ▪ Utilizar gel hidroalcohólico para lavarse las manos frecuentemente. ▪ Mantener distancia interpersonal de 1.5 metros.
Plan de contingencia	Seguir las indicaciones del personal sanitario para una correcta recuperación de la salud.
Probabilidad	**Probable.** La tasa de infección está siendo muy alta, sobre todo en personas jóvenes.
Impacto	**Muy alto.** Los testimonios de las personas a mi alrededor que lo han padecido nos hace una media de 3 semanas para una completa recuperación, pero varía de persona a persona.

Tabla 2.29: RP1. Contagiarse de COVID-19

RP2. Otras enfermedades	
Descripción	Si bien es cierto que a día de hoy el foco está en la pandemia COVID-19, se pueden contraer otras enfermedades que afecten al progreso del desarrollo.
Prevención	Siguiendo las medidas de RP1 se eliminan probablemente de la ecuación las típicas enfermedades estacionales como la gripe. Con otro tipo de enfermedades la incertidumbre es demasiado grande para poder hacer una prevención sólida.
Plan de contingencia	Seguir las indicaciones del personal sanitario para lograr una correcta recuperación.
Probabilidad	**Improbable.** Una de las consecuencias de que todos usemos mascarilla es que la transmisión de las enfermedades ha disminuido mucho.
Impacto	**Medio.** Dependiendo de la enfermad pero se estima una media de 3 días de inactividad.

Tabla 2.30: RP2. Otras enfermedades

RP3. Poco tiempo disponible	
Descripción	Estudiar y trabajar a la vez deja muy poco tiempo disponible para dedicarle al proyecto. El cansancio puede provocar que no se tengan la energía suficiente para continuar el desarrollo.
Prevención	▪ Gestionar el tiempo del día en un calendario y respetar lo que se ha decidido hacer. ▪ Planificar descansos para que no se produzca *burnout*.
Plan de contingencia	Si se pierde demasiado tiempo será necesario sacrificar tiempo libre para ponerse al día con el proyecto.
Probabilidad	**Muy probable.**
Impacto	**Muy alto.** En el peor de los casos, en un periodo de exámenes se estima que puede haber un retraso de 4 semanas.

Tabla 2.31: RP3. Poco tiempo disponible

RH1. Avería del equipo de trabajo	
Descripción	El equipo de trabajo puede sufrir una avería debido a varios factores y es posible que se pierda el código del proyecto.
Prevención	▪ Utilizar un control de versiones que sincronice los cambios en un repositorio online, para tener siempre una copia de seguridad. ▪ Realizar backups frecuentes al repositorio, con cada cambio. ▪ No comer ni beber cerca del equipo de trabajo. ▪ No dejar el equipo de trabajo en lugares de los que se pueda caer y romperse.
Plan de contingencia	Recuperar las copias de seguridad y utilizar otro equipo de trabajo.
Probabilidad	**Poco probable.**
Impacto	**Muy alto.** Perder los datos significaría que no se puede continuar el desarrollo. El impacto sería crítico.

Tabla 2.32: RH1. Avería del equipo de trabajo

RH2. Robo del equipo de trabajo	
Descripción	Sobre todo en lugares públicos, dejar el portátil sin vigilancia puede desembocar en un robo del mismo. Si no se han realizado copias de seguridad, se perdería por completo el proyecto.
Prevención	▪ Nunca dejar el portátil sin vigilancia en un lugar público. ▪ Utilizar un control de versiones que sincronice los cambios en un repositorio online, para tener siempre una copia de seguridad. ▪ Realizar backups frecuentes al repositorio, con cada cambio.
Plan de contingencia	Recuperar las copias de seguridad y utilizar otro equipo de trabajo.
Probabilidad	**Poco probable.**
Impacto	**Muy alto.** Perder los datos significaría que no se puede continuar el desarrollo. El impacto sería crítico.

Tabla 2.33: RH2. Robo del equipo de trabajo

RD1. Perfeccionismo innecesario	
Descripción	El perfeccionismo puede llevar a alargar una tarea demasiado tiempo o incluso a la parálisis completa del trabajo por percibir que no se está haciendo en las condiciones *ideales*. El perfeccionismo y la procrastinación van unidos de la mano.
Prevención	▪ Respetar las horas que se han estimado. Si ya se ha conseguido un *producto mínimo viable* en el tiempo acordado se dará por terminada y se pasará a la siguiente. ▪ Ser consciente de que las condiciones ideales no existen y que es mejor tener algo hecho medio bien que no tener nada hecho por no ser perfecto.
Plan de contingencia	Se recuperará tiempo quitándoselo a otras tareas si es posible, pero esto provocará que la calidad de ellas sea menor al esperado.
Probabilidad	**Probable.**
Impacto	**Alto.** Un perfeccionismo excesivo podría llevar a reimplementar una y otra vez funcionalidades que podrían llevar hasta 3 semanas de retraso.

Tabla 2.34: RD1. Perfeccionismo innecesario

RD2. Errores en la planificación	
Descripción	Si se estiman mal las tareas se provocará un aumento del tiempo total necesario para completar el desarrollo.
Prevención	▪ Hacer un análisis con calma, estudiando las posibles dificultades que serán necesarias hacer frente para terminar la tarea. ▪ Ser consciente de las limitaciones de uno mismo, será necesario aprender muchas cosas nuevas para completar el proyecto.
Plan de contingencia	Se recuperará tiempo quitándoselo a otras tareas si es posible, pero esto provocará que la calidad de ellas sea menor al esperado.
Probabilidad	**Probable.**
Impacto	**Alto.** 2 semanas de retraso.

Tabla 2.35: RD2. Errores en la planificación

RD3. Bugs en el motor de desarrollo	
Descripción	Un *bug* es un error en el funcionamiento de un software. Es posible que ciertas tareas que se quieran implementar no funcionen correctamente por un error en el propio motor de desarrollo.
Prevención	▪ Investigar en Internet si es posible realizar la tarea de la forma en la que se ha pensado hacerla. ▪ Estar al día de las noticias del motor de desarrollo de juegos seleccionado para saber si hay algún bug crítico. ▪ Actualizar a la última versión del motor, ya que tendrá las últimas correcciones.
Plan de contingencia	Se pedirá ayuda en el foro del motor de desarrollo.
Probabilidad	**Improbable.**
Impacto	**Medio.** Posiblemente se pueda realizar la tarea si se busca otra forma de implementarla. Se estima que el retraso pueda ser de 1 semana.

Tabla 2.36: RD3. Bugs en el motor de desarrollo

2.4.1. Plan de seguimiento de riesgos

Priorización de los riesgos definidos

A continuación se muestran, de mayor a menor importancia, los riesgos que se han identificado.

1. **RH1. Avería del equipo de trabajo**: riesgo de mayor relevancia ya que si bien el efecto es el mismo que el de sufrir un robo, es más probable que este ocurra.
2. **RH2. Robo del equipo de trabajo**: la consecuencia produce un impacto crítico, por lo tanto es el segundo más importante.
3. **RP3. Poco tiempo disponible**: de los riesgos que no provocan un impacto crítico es el de mayor importancia por el tiempo de retraso que provocaría.
4. **RP1. Contagiarse de COVID-19**: riesgo de importancia media pero probable que ocurra.
5. **RD1. Perfeccionismo innecesario**: es un riesgo que se puede evitar si se respeta el plan de prevención, por lo tanto tiene una importancia menor.
6. **RD2. Errores en la planificación**: de importancia media baja ya que no produce tanto retraso en las fechas del proyecto.
7. **RD3. Bugs en el motor de desarrollo**: es poco probable, y el impacto es menor.
8. **RP2. Otras enfermedades**: riesgo menos prioritario debido a que es prácticamente imposible de prever.

Planificación del seguimiento

Es de vital importancia que para evitar los riesgos RH1 y RH2 se hagan copias de seguridad cada vez que se produzca un cambio en el proyecto.

Para los riesgos RP3, RD1 y RD2 el seguimiento será semanal. Consistirá en revisar el tiempo empleado en las tareas y el tiempo estimado, de esa manera

se podrán identificar temprano los retrasos que se pudieran estar produciendo y remediarlo.

Por último, para los riesgos que tienen que ver con la salud, que son RP1 y RP2, el seguimiento consistirá en prestar atención a los síntomas y contactar con el profesional sanitario cuanto antes para recibir un pronto diagnóstico.

2.5. Evaluación económica

Si se hubiera elegido un motor de juego como *Unreal Engine* probablemente se hubiera requerido una máquina con muchos más recursos. El portátil que se ha elegido para el desarrollo, *ASUS GL552VW*, es un portátil de gama media que cumple las exigencias del tipo de juego que se va a realizar. Como se va a utilizar el motor de desarrollo de videojuegos *Godot Engine* para realizar un juego simple de dos dimensiones, no se necesita una fuerza de computación grande. El importe es de 860€ con una amortización a 5 años. Se ha calculado la amortización mensual, y por 7 meses de uso al 20 % quedaría:

$$((860/5)/12) * 7 * 20\,\% = 20,07€$$

Entre los gastos únicos se ha añadido *Aseprite*, el programa de dibujo digital de *Pixel Art*[2] con el que se dibujarán los gráficos del videojuego. Es importe de la licencia asciende a 16,79€.

Gastos asociados al proyecto en €	
Descripción	**Total**
ASUS GL552VW	20,07€
Aseprite	16,79€
Salario (22€/hora)	278 * 22 = 6116 €
TOTAL	**6152,86**

Tabla 2.37: Gastos asociados al proyecto.

Como se puede observar, el coste total del proyecto ascenderá a 6152,86 €.

2.5.1. Recuperación de la inversión

Para recuperar la inversión, una de las opciones que se ha valorado es publicar el juego en la tienda de videojuegos Steam. Para publicarlo hay que hacer una inversión de 100€ por juego y con cada venta hay que tener en cuenta que Steam

[2]Pixel Art o Arte Pixelado es un diseño artístico creado de forma digital basado en píxeles.

se lleva el 30 %. Por lo tanto, si las ganancias son el 70 % por unidad vendida, si el precio es de 6€/unidad[3]:

$$\frac{6252,86€}{6\frac{€}{u} * 70\,\%} = 1489u$$

Es decir, habría que vender 1489 copias en Steam para recuperar la inversión, a partir de la copia 1490 se empezarían a generar beneficios. Además, si el juego fuera lo suficientemente popular, también se podría valorar la venta de *merchandising*.

[3]Este precio se ha decidido en base al alcance del juego comparándolo con otros títulos similares. El contenido que ofrece este TFG no justificaría un precio mayor.

Capítulo 3

Antecedentes

3.1. Situación Actual

Cuando propuse este trabajo a mi tutor, creé un pequeño prototipo en el que puse a un personaje en una plataforma junto a un árbol y a un mineral. El personaje se podía mover y era capaz de talar los árboles.

Figura 3.1: Imagen del prototipo inicial.

Expliqué que el objetivo sería crear un videojuego en el que los niveles por los que se moviera el personaje fueran creados proceduralmente. Además, tendría que luchar contra enemigos para poder conseguir mejor equipamiento y así derrotar al jefe final.

El prototipo había sido creado con un motor de desarrollo de videojuegos llamado *Godot Engine* pero cuando se aceptó la proposición de este TFG había que decidir si continuar usándolo o elegir otro que quizás cumpliera mejor con las exigencias del proyecto.

3.2. Decidir el motor de desarrollo

Un motor de juego es un sistema diseñado para la creación de videojuegos que proporciona un conjunto de funciones que suelen ser habituales para llevar a cabo el desarrollo. La función principal es dotar al juego de un motor gráfico para el renderizado de los modelos y animaciones que forman el videojuego. A menudo, los motores también incorporan un entorno de desarrollo formado por varias herramientas para facilitar a los desarrolladores el trabajo, como un motor de físicas o un motor de colisiones. [1]

Si bien es cierto que contar con un motor de juego propio desarrollado por uno mismo tiene sus ventajas, como una mayor personalización u optimización, desarrollar uno desde cero es una tarea que consume una gran cantidad de tiempo. Por lo tanto, una práctica habitual es utilizar motores ya desarrollados por terceros para ahorrarse este paso y así poder iniciar inmediatamente el desarrollo del videojuego. Eso es lo que he decidido hacer para este proyecto.

Algunos ejemplos de los motores de juego más usados a día de hoy son Unity, Unreal Engine o Godot. Los campos que se han decidido valorar han sido el lenguaje de programación, precio, la plataforma desde la que se puede desarrollar, los requerimientos del sistema y la cantidad de tutoriales que hay en Internet.

3.2.1. Unity

Figura 3.2: Logotipo de Unity.

Si hablamos de motores, no podríamos empezar por otro que no fuera el líder de ellos, Unity[4]. A día de hoy es uno de los más usados del mundo y aparte de juegos en tres dimensiones, también se han realizado multitud de juegos de plataformas en dos dimensiones con él como el que se pretende hacer.

- **Lenguaje de programación:** C#.
- **Precio:** Su uso es gratuito hasta que se alcanzan los 100.000$ en ingresos durante los últimos 12 meses.
- **Plataforma:** Windows.
- **Requerimientos del sistema:** Medios.
- **Tutoriales:** Gran cantidad de tutoriales.

3.2.2. Unreal Engine

Figura 3.3: Logotipo de Unreal Engine.

Unreal Engine es el motor de videojuegos más potente que existe para realizar videojuegos en tres dimensiones. Su capacidad es tal que hasta se usa en Hollywood para hacer efectos especiales.

- **Lenguaje de programación:** C++ y Blueprints (método para definir la lógica sin programar).
- **Precio:** 5 % de los royalties cuando se monetice el juego y los ingresos hayan alcanzado 1.000.000$.
- **Plataforma:** Windows.
- **Requerimientos del sistema:** Altos.
- **Tutoriales:** Gran cantidad de tutoriales.

3.2.3. Godot

Figura 3.4: Logotipo de Godot Engine.

Es un motor de código abierto que se ha hecho muy popular últimamente al ofrecer un entorno de trabajo muy intuitivo en el que los componentes se organizan por *Nodos*. Su capacidad para desarrollo en tres dimensiones es limitada pero en dos dimensiones está a la par con Unity.

- **Lenguaje de programación:** GDScript y C#.
- **Precio:** Totalmente gratuito.
- **Plataforma:** Windows/Mac/Linux.
- **Requerimientos del sistema:** Bajos.
- **Tutoriales:** Poca cantidad de tutoriales.

3.2.4. Decisión

Como se quiere desarrollar un videojuego en dos dimensiones se descarta Unreal Engine. Para dos dimensiones la decisión estaría entre Unity y Godot Engine. Ya que uno de los objetivos es utilizar herramientas de código abierto, Godot sería una buena elección, por su licencia MIT y porque permite un desarrollo desde Linux. Su lenguaje de desarrollo GDScript es similar a Python así que ya dispongo de unos conocimientos básicos de desarrollo y la organización del proyecto mediante nodos es muy intuitiva para los que desarrollamos con programación orientada a objetos. Por lo tanto, se va a utilizar Godot aunque haya menor cantidad de tutoriales. Si se encuentran problemas se acudirá a la comunidad para intentar resolver las dudas.

3.3. Generación procedural de contenido

Cada partida a este videojuego debe generar una estructura del nivel diferente a la partida anterior. Para ello, se van a estudiar diferentes métodos que se utilizan en otros videojuegos.

3.3.1. Perlin noise

Uno de los algoritmos típicos a la hora de generar contenido de forma procedural es el ruido Perlin. Un algoritmo que dado un conjunto de parámetros produce una serie de números del 0,0 al 1,0. Si se interpretase una escala de grises siendo 0 negro y 1 blanco se podría dibujar una imagen como la siguiente:

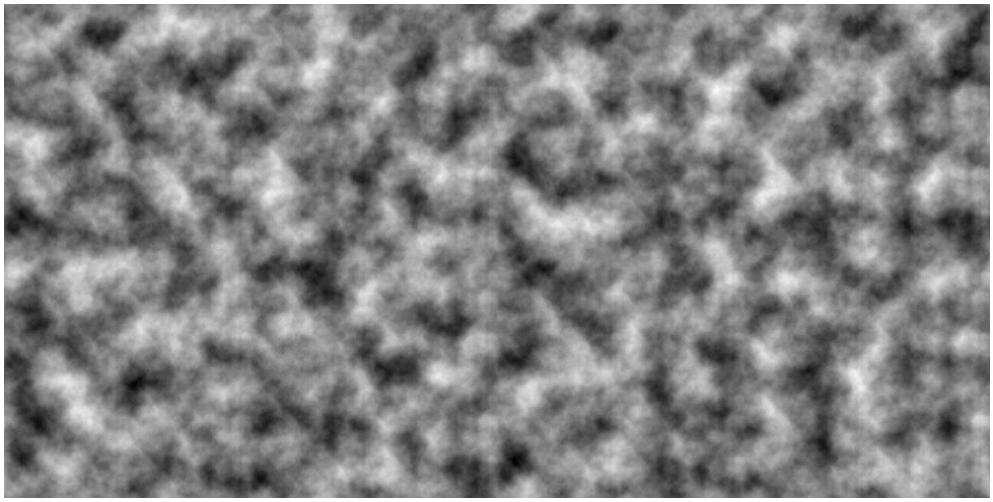

Figura 3.5: Representación del ruido Perlin.

Los parámetros que se le dan al algoritmo determinarían la cantidad de blanco sobre negro, lo agrupados que estarían los puntos blancos entre sí, la escala, etc.

La utilización es muy sencilla, ya que viene incluido en la mayoría de motores de desarrollo de videojuegos. Se le pasa como parámetro la coordenada y el algoritmo devolvería un *float* del 0,0 al 1,0. Para asegurarse que la generación inicial siempre es aleatoria, se inicia con una *semilla* aleatoria. Es decir, el número inicial que usará para empezar a generar sus resultados. Si se le da la misma semilla siempre generará el mismo output.

Minecraft

En el juego Minecraft se utiliza para la generación de montañas y cuevas. Como puede apreciarse en la siguiente imagen, si nos imaginamos la escala de grises como la altura del suelo se generaría lo siguiente:

Figura 3.6: Montañas en Minecraft.

Terraria

Terraria es un juego en dos dimensiones más similar al proyecto que se quiere realizar que Minecraft. Aquí también se utiliza para la generación de cuevas, pero no se utiliza la escala de grises para la altitud si no que se utiliza para crear caminos bajo tierra.

Figura 3.7: Cuevas en Terraria.

3.3.2. Plantillas

Este método no tiene nada que ver con el anterior. Se utiliza cuando se quiere tener un mayor control de los tipos de estructuras que se quieren generar.

Spelunky

El juego Spelunky utiliza plantillas de texto para generar proceduralmente sus niveles [3]. Cada letra representa un tipo de elemento en el juego (bloque, probabilidad de que haya enemigo, escaleras...etc) y juntando diferentes plantillas aleatoriamente una junto a otra se va generando el nivel. Aquí tenemos el ejemplo de una plantilla:

```
1 1000000000
2 1000000000
3 1000000000
4 1000000000
5 1001111100
```

```
1001222100
1002222000
1111111111
```

Si a la hora de dibujar el mapa se interprentan los 0 como espacio libre y los 1 como bloque sólido se construiría algo aproximado a la primera sala de la imagen a continuación. Los 2 podrían ser bloques sólidos que no tendrían un 100 % de probabilidad de aparecer. En resumen, dependiendo de la interpretación que se le quiera dar a la letra o número de la plantilla se generará algo específico en el mapa.

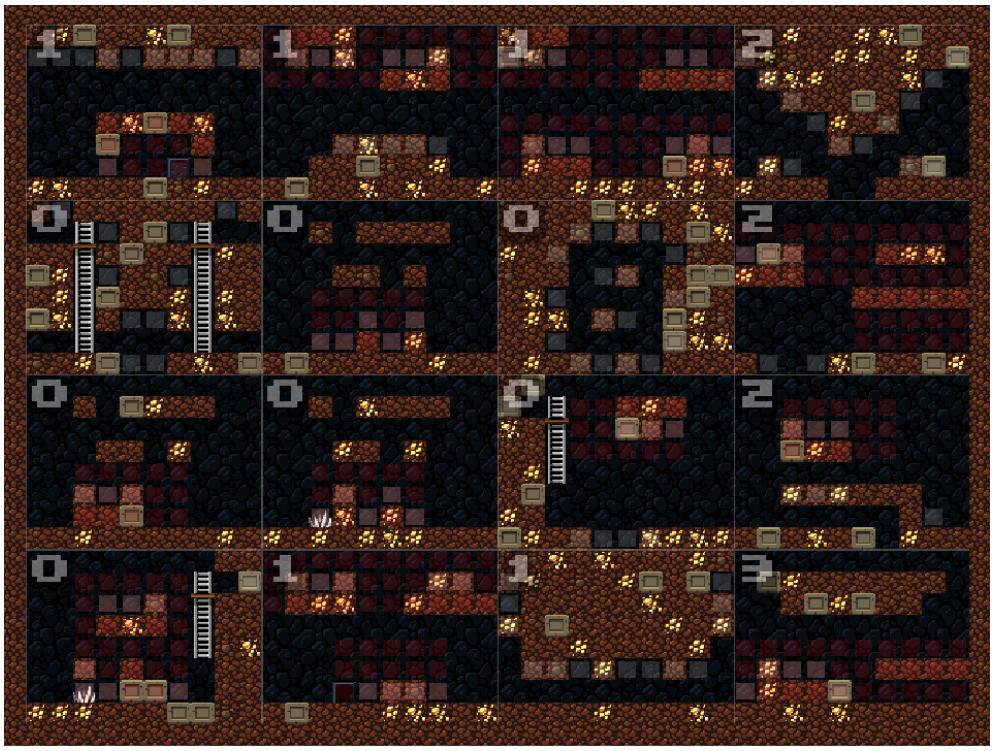

Figura 3.8: Conexión de salas en Spelunky.

Caveblazers

Otro ejemplo de un juego que combina las plantillas aleatoriamente para generar las estructuras del nivel.

Figura 3.9: Cuevas en Caveblazers.

3.3.3. Decisión

La generación de niveles mediante plantillas aporta un mayor control del tipo de nivel que se quiere crear. Es menos aleatorio que el ruido Perlin y dependiendo de la plantilla, da un aspecto más ordenado al nivel. Puedo imaginarme perfectamente usando este método para generar las plataformas, el suelo, los recursos que se pueden recolectar, los enemigos con los que luchar, etc.

Sin lugar a dudas, creo que es el método adecuado para este proyecto y es el que se intentará implementar.

Capítulo 4

Captura de requisitos

En este capítulo se detallan los requerimientos que tiene que cumplir el proyecto que se va a desarrollar. Se van a mostrar en diagramas de casos de uso las necesidades de los actores y se va a explicar cada caso uno de ellos.

4.1. Jerarquía de actores

Solo hay un agente externo al sistema "juego", el jugador. Por lo tanto, no hay una jerarquía de la que otros actores puedan heredar funcionalidades, solo tendremos al jugador.

4.2. Diagramas de casos de uso

He decidido separar los casos de uso en dos imágenes diferentes para que se distingan mejor sobre el papel. Los he separado en los casos que están relacionados con el movimiento del jugador sobre el nivel y los que no lo están.

4.2.1. Casos de uso no relacionados al movimiento del personaje

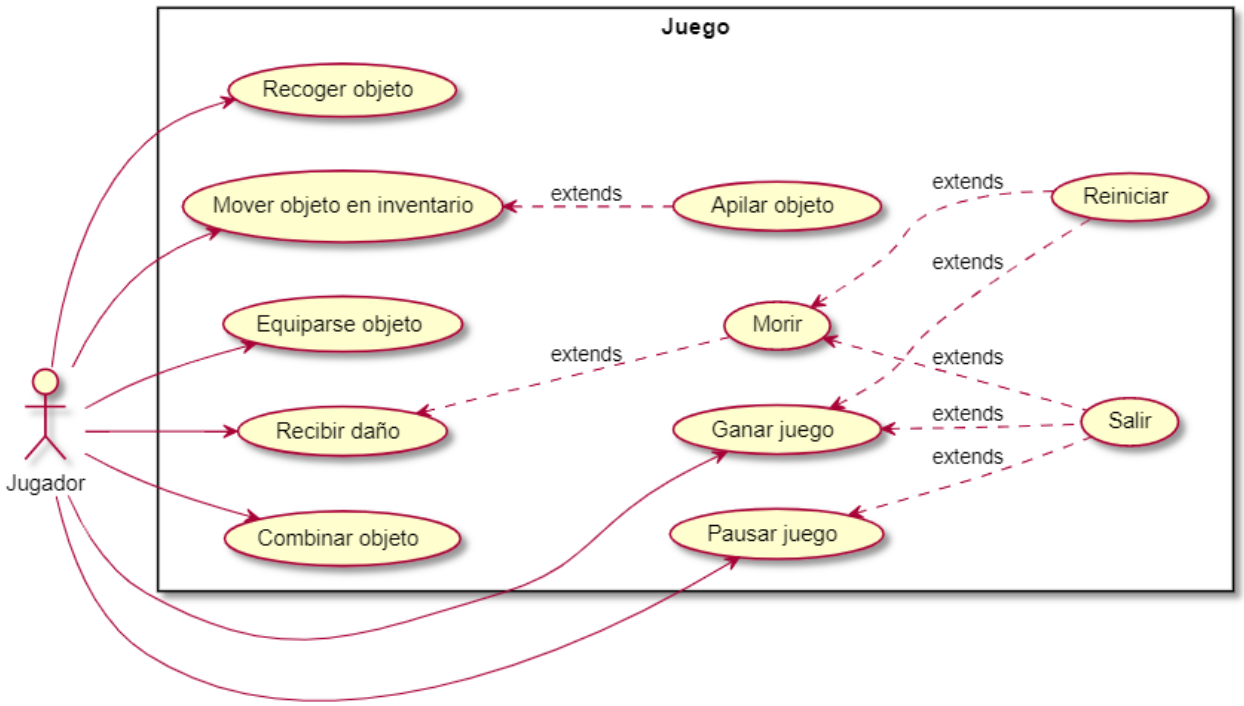

Figura 4.1: Casos de uso parte 1.

- **Pausar juego:** el jugador debe ser capaz de pausar el juego.
- **Ganar juego:** cuando el jugador cumpla la condición para ganar la partida "Vencer al jefe final", se debe mostrar al jugador que ha terminado la partida.
- **Recibir daño:** debe ser capaz de recibir daño de los enemigos.
- **Morir:** si se ha recibido el suficiente daño y los puntos de salud bajan a una cantidad menor o igual a cero, el jugador morirá.
- **Mover objeto en inventario:** usando el ratón, el jugador tiene que ser capaz de reordenar los objetos que se encuentran en su inventario.

- **Equiparse objeto:** ya sea armadura o el arma a usar, el jugador debe poder equiparse el objeto deseado. Esto provocará que sus estadísticas de combate mejoren o empeoren dependiendo del objeto.
- **Recoger objeto:** se podrán recoger objetos del suelo cuando se camine cerca de ellos, lo que hará que aparezcan en el inventario del jugador.
- **Apilar objeto:** cuando dos objetos sean del mismo tipo en el inventario y estos sean apilables, el jugador debe ser capaz de juntar los objetos para que solo ocupen una casilla en el inventario.
- **Combinar objeto:** cuando se dispongan los materiales necesarios para crear objetos, la lista debe mostrarse al jugador para que pueda seleccionar el que quiera combinar. Esto provocará que desaparezcan los objetos de la receta de combinación del inventario y que aparezca el objeto combinado.
- **Reiniciar:** debe ser capaz de reiniciar el juego tanto si muere como si gana la partida.
- **Salir:** será capaz de salir del juego tanto si muere, gana o pausa el juego.

4.2.2. Casos de uso relacionados al movimiento del personaje

Como muchos casos de uso se pueden realizar mientras se está ejecutando otro, por ejemplo moverse mientras se ataca, se han indicado esas relaciones en el diagrama.

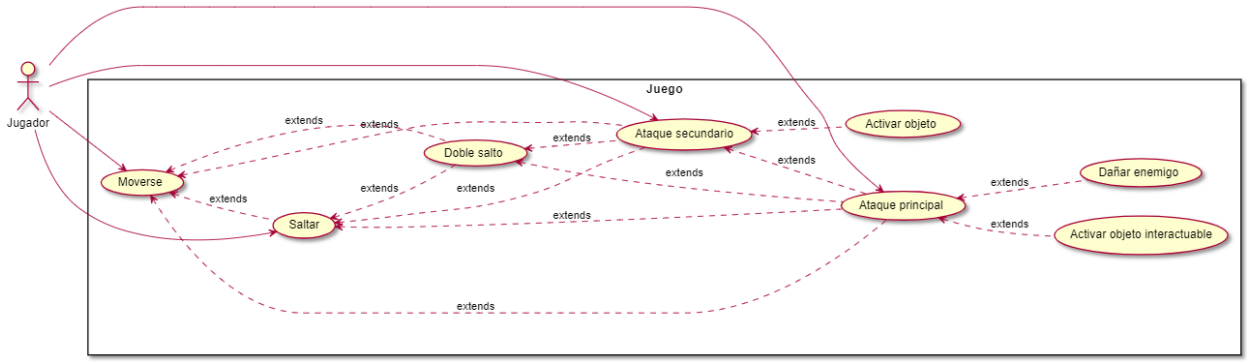

Figura 4.2: Casos de uso parte 2.

- **Moverse:** el jugador puede usar las teclas [A] y [D] para moverse a la izquierda o derecha respectivamente.
- **Saltar:** pulas la tecla [Espacio] hará que el jugador salte.
- **Doble salto:** pulsar de nuevo [Espacio] cuando el jugador se encuentre en el aire provocará un segundo salto antes de aterrizar.
- **Ataque principal:** el hacer click izquierdo del ratón provoca un golpe con el objeto que se tenga equipado en la mano. Si no hay objeto se pega un manotazo.
- **Ataque secundario:** click derecho para intentar hacer un ataque secundario con el objeto que se tiene en la mano. Si el objeto no tiene una acción secundaria se hará un ataque principal.
- **Activar objeto:** si el objeto poseía una acción secundaria, como puede ser una poción, se activará el objeto. En este caso se consumirá y se aplicará el efecto correspondiente.
- **Dañar enemigo:** los ataques pueden provocar daño a los enemigos.
- **Activar objeto interactuable:** las acciones de ataque pueden activar objetos interactuables. Si se tiene un hacha en la mano y se ataca a un árbol, este se talará y soltará madera en el mapa. Usar el pico da la capacidad de minar y hay objetos que son interactuables con cualquier objeto como las setas, que podrán ser recogidas del suelo directamente.

Capítulo 5

Análisis y diseño

El proyecto se ha dividido en 5 prototipos (PT). Cada uno de ellos representa un incremento de funcionalidades respecto al anterior. La razón por la que se ha realizado esta división es porque se desea pasar por una fase de pruebas en cada uno de ellos, en lugar de esperar a terminar toda la implementación para hacerla.

Es importante que no se mezclen errores de fases tempranas del proyecto con errores de las últimas fases, pues aumentaría la dificultad para encontrar soluciones al no estar bien probados los sistemas que están interconectados.

A continuación se va a ir explicando prototipo a prototipo el diseño que se ha decidido hacer.

5.1. PT1. Creación de niveles

Este primer prototipo consiste en añadir la lógica de generación procedural del nivel. Esto quiere decir que al finalizar este prototipo, al iniciar la partida el personaje aparecerá en un nivel de plataformas que debido a la aleatoriedad introducida, será diferente cada vez.

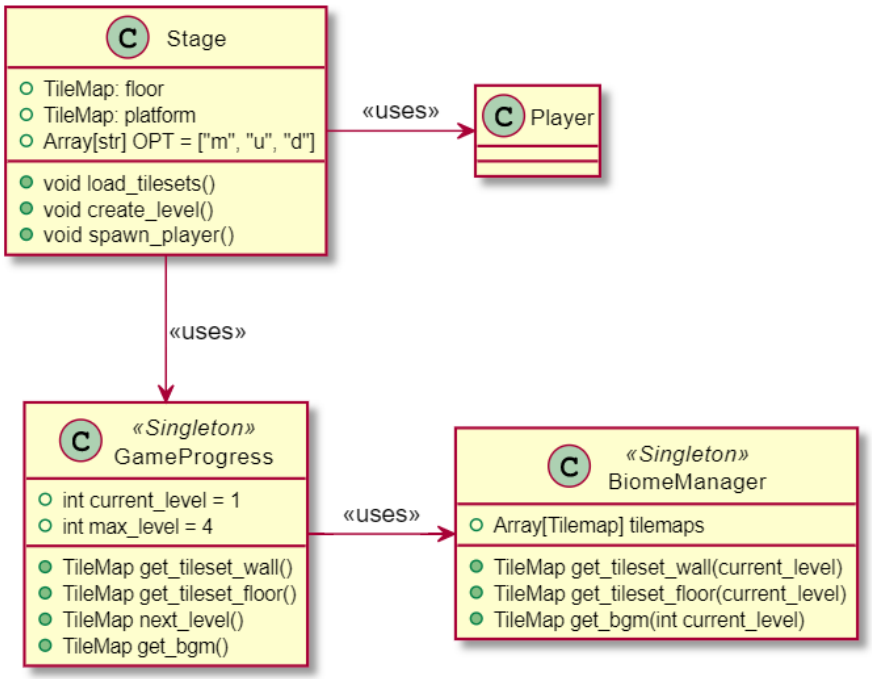

Figura 5.1: PT1. Diagrama de clases

- **Stage:** clase que representa al mapa. Su función es la de crear el mapa y poner al jugador en él.
- **Player:** representa al jugador, ya se dispone de una clase básica del jugador que poner en el mapa.
- **GameProgress:** clase única (singleton) que se encarga de saber en que punto de progreso está la partida. Actúa como una fachada que conectará diferentes sistemas.
- **BiomeManager:** guarda toda la información de los biomas, es decir, las diferentes pantallas que se van a crear (bosque verde, bosque de champiñón, castillo). La idea es, aparte de guardar los mapas, usar esta clase para almacenar en el futuro los enemigos que estarán en cada nivel.

La creación del nivel se haría de la siguiente manera:

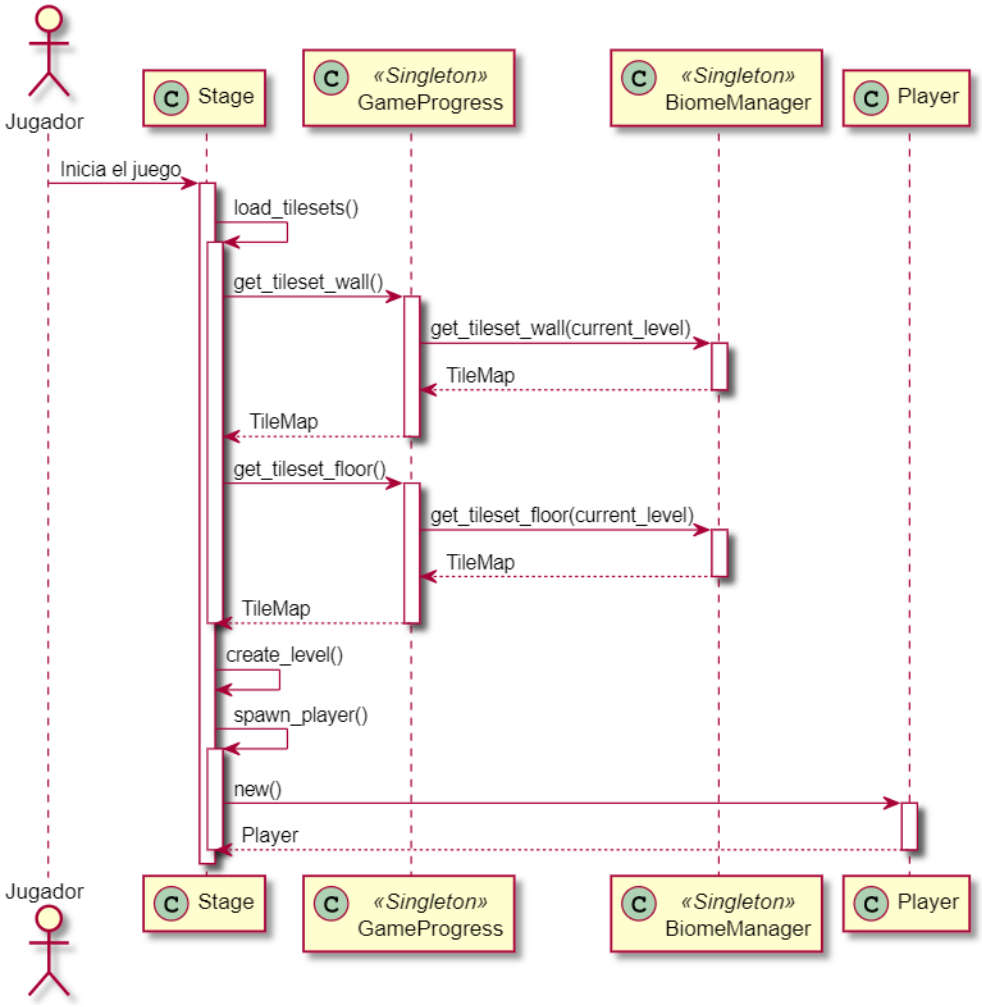

Figura 5.2: PT1. Diagrama de secuencia de la creación del nivel.

Lo más complicado, y que llevará más tiempo, va a ser implementar la función *create_level* de la clase *Stage*. Habrá que definir la lógica de cómo se van a leer las plantillas y de cómo se conectarán entre sí para interconectar las diferentes salas. Hacer que encajen entre ellas va a ser un reto.

5.2. PT2. Elementos interactuables

Consiste en añadir a la generación de nivel elementos con los que el jugador podrá interactuar. Por ejemplo árboles que se podrán cortar, plantas que recolectar y minerales que minar.

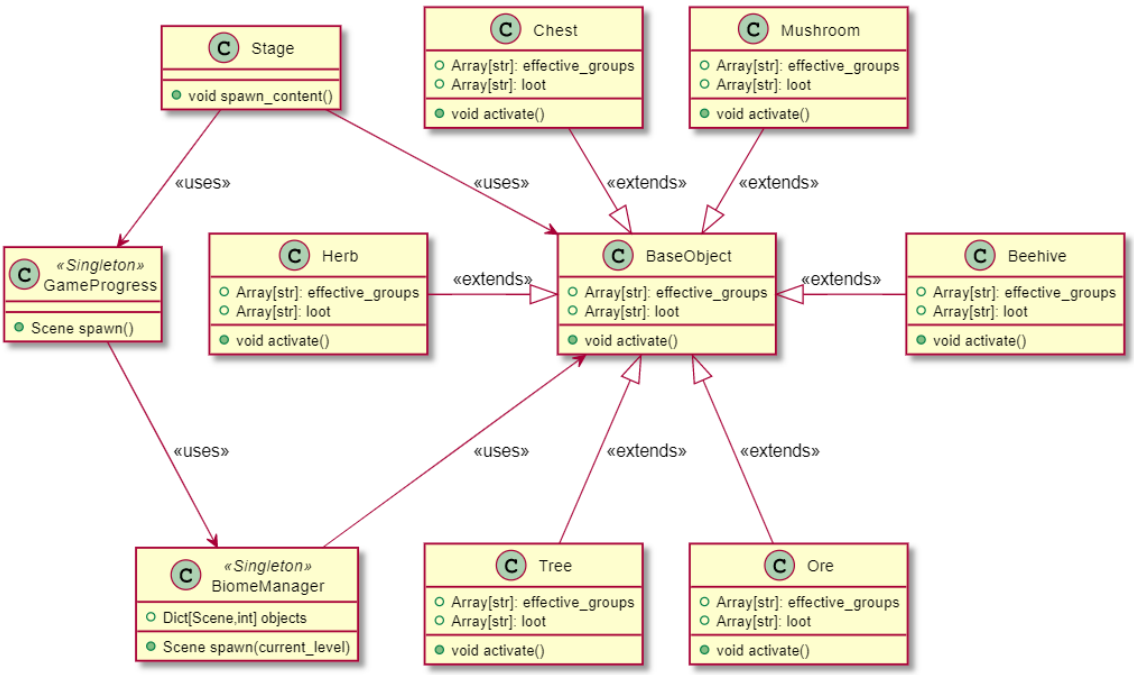

Figura 5.3: PT2. Diagrama de clases.

- **Stage:** se ha añadido una nueva función que sirve para pedir un objeto interactuable a GameProgress. Sigue la misma lógica que cuando se pedían los TileMaps en PT1.
- **BaseObject:** clase base desde la que se van a crear los demás elementos. Todos contienen una lista de *effective_groups*, es decir, una lista de a que tipo tiene que pertenecer lo que interactúe con ello. Por ejemplo, la clase *Tree* tiene como *effective_groups* solamente a *axe* (hacha), por lo tanto, solamente un objeto que pertenezca al grupo *axe* provocará que se ejecute su función *activate*.
- **Elementos que extienden BaseObject:** ore (mineral), chest (cofre), mushroom (seta), herb (hierba), beehive (colmena), tree (árbol). Cada uno

reimplementará la función *activate* para que produzcan sus propios efectos cuando se interactúe con ellos. Tienen una lista de *loot* (botín), es decir, los objetos que soltarán cuando se haya interactuado con ellos.

A continuación se muestra de forma más visual la lógica de los *effective_groups*.

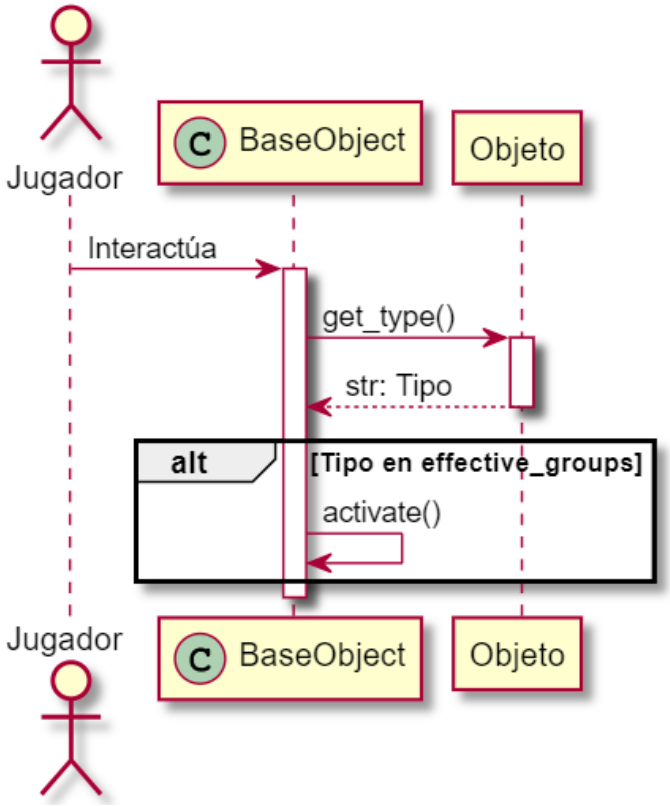

Figura 5.4: PT2. Diagrama de secuencia de la activación de los objetos interactuables.

5.3. PT3. Implementación de interfaces

Consiste en añadir la interfaz que muestre la vida del jugador, el inventario de los objetos en su posesión, las combinaciones posibles entre sus objetos [1] y los menús necesarios para empezar a jugar, pausar y terminar el juego.

Aparte de las nuevas interfaces, lo que más destacaría es la creación de 3 clases nuevas que almacenarán información de los objetos:

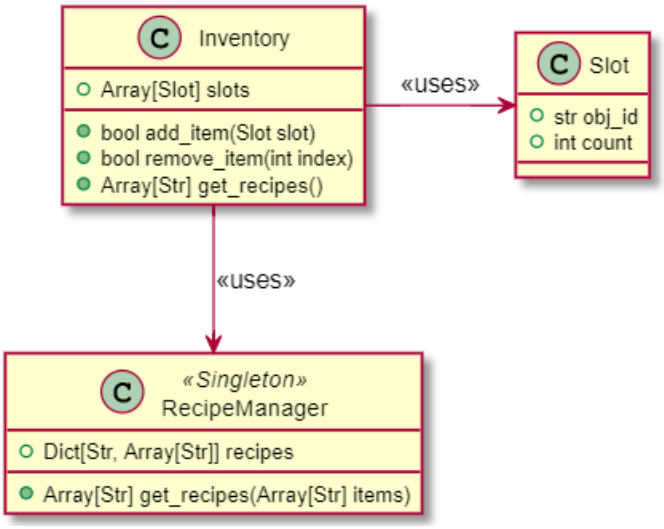

Figura 5.5: PT3. Diagrama de clases que tienen que ver con el inventario.

- **Slot:** representa una casilla del inventario.
- **Inventory:** conjunto de *Slots*, contiene la lógica para añadir o eliminar slots.
- **RecipeManager:** dado un conjunto de objecos, devuelve una lista de los elementos que se pueden crear.

[1] Un ejemplo de combinación de objetos podría ser combinar una hierba con miel para producir una poción con la que el jugador podrá restaurar parte de su salud.

5.4. PT4. Objetos utilizables

Ahora que ya se dispone de una interfaz para visualizar objetos, se implementarán los objetos que el jugador podrá utilizar. Esto incluye herramientas para talar árboles, minar minerales o dañar enemigos, poción para restaurar salud y armaduras.

En este caso no se ha precisado de una base de datos, ya que la información de los objetos se ha decidido guardar en un simple fichero *JSON* que cargará el singleton *ObjectInfo*.

Al principio se había pensado en implementar una clase para cada objeto, pero si simplemente se cambia la imagen y el identificador del objeto ya debería ser suficiente. De tal forma que con una sola clase se pueden representar todos los objetos.

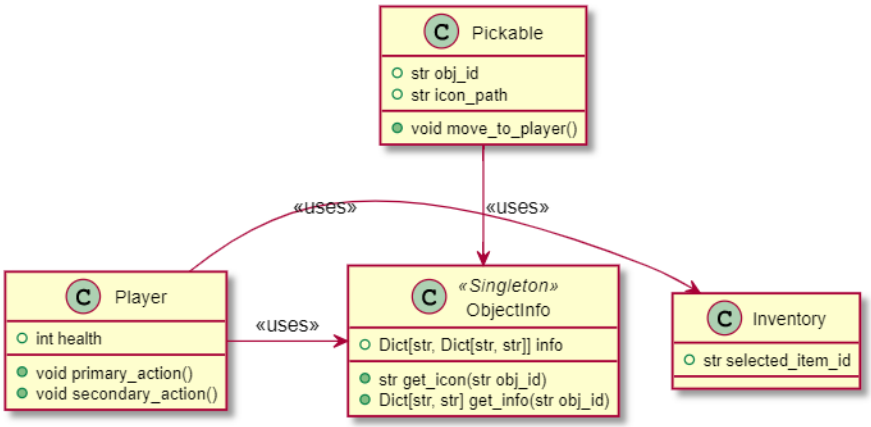

Figura 5.6: PT4. Diagrama de clases para utilizar objetos.

- **Pickable:** representa un objeto que se puede coger del suelo. Utiliza a *ObjectInfo* para saber que icono mostrar en el juego.
- **Inventory:** ahora tiene que tener en cuenta el objeto que está seleccionado, para que el jugador pueda extraer esa información.
- **Player:** el jugador puede realizar la acción primaria o secundaria del objeto dependiendo si hace click izquierdo o derecho del ratón. Utiliza a *Inventory* para saber que objeto lleva equipado y para añadir los nuevos objetos que

recoge del suelo. Para saber que tiene que hacer con el objeto obtiene la información de *ObjectInfo*, esto permite que añadir objetos o modificarlos se haga solamente desde este singleton.

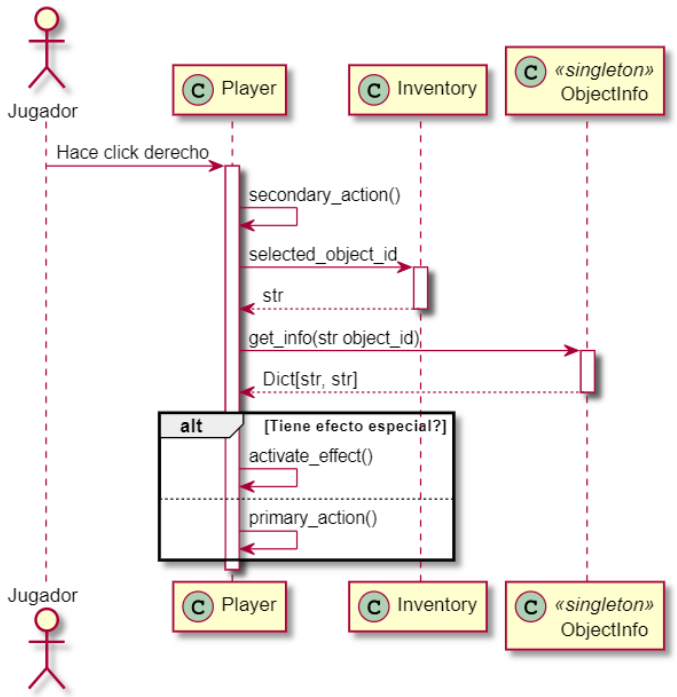

Figura 5.7: PT4. Diagrama de secuencia de activación de un objeto al hacer click derecho.

5.5. PT5. Creación de enemigos

Última fase del proyecto. Se añadirán enemigos que atacarán al personaje y que al ser debilitados soltarán objetos que el jugador podrá utilizar.

El diseño inicial de las clases era el siguiente:

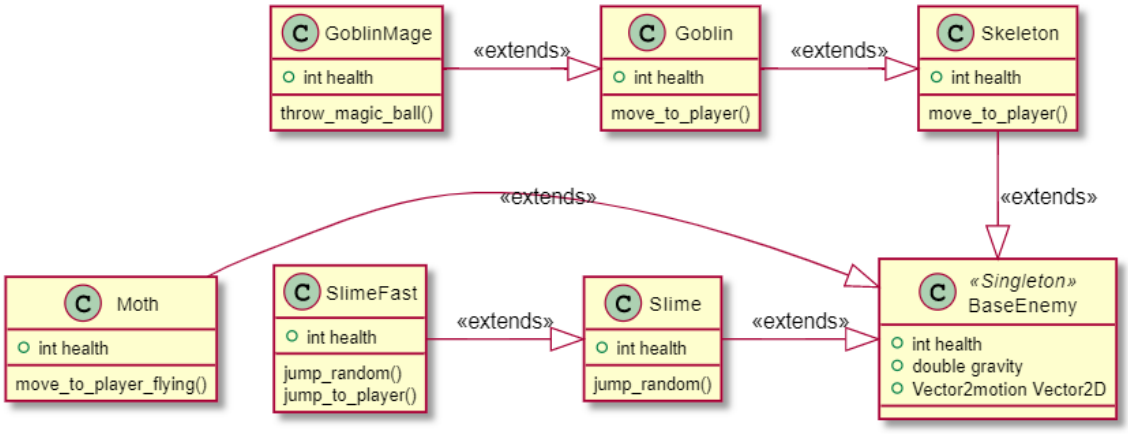

Figura 5.8: PT5. Diagrama inicial de clases de enemigos.

- **Goblin**: duende.
- **GoblinMage**: duende mago.
- **Skeleton**: esqueleto.
- **Moth**: polilla.
- **Slime**: babosa.
- **SlimeFast**: babosa rápida.

A primera vista puede parecer un diseño inicial válido, ya que cumpliría los requisitos de lo que tienen que hacer los enemigos. Sin embargo, hay un problema que no se ve a simple vista: es un sistema demasiado rígido. Los enemigos dependen demasiado entre sí.

¿Qué pasaría si cambian los requisitos y quisiéramos que el Skeleton saltase como lo hacen los Slimes? Habría que asegurarse que los enemigos que dependen de él no se vieran afectados por los cambios. ¿O que pasaría si quisiéramos que los Slime usen magia como el GoblinMage? No habría forma de reutilizar funcionalidades. Es resumen, cada cambio en un enemigo podría afectar en cascada al resto y extender el número de enemigos sería cada vez más difícil. Por lo tanto, es un mal diseño.

5.5.1. Herencia VS Composición

Uno de los principios SOLID es favorecer la composición sobre la herencia. Esto quiere decir que es preferible que las clases contengan instancias de otras clases que implementan una funcionalidad específica, en lugar de heredar esa funcionalidad mediante la herencia.

¿Y si en lugar de implementar la funcionalidad directamente en la clase del enemigo y reusarla mediante la herencia se sacara absolutamente todo aparte?

A continuación se muestra una tabla en la que se han detallado los diferentes comportamientos que construyen a los enemigos.

Componente	SK	GB	GM	MT	SM	SF	OG
PlayerDetector	x	x	x	x	-	x	x
AnimationPlayer	x	x	x	x	x	x	x
CanDie	x	x	x	x	x	x	x
DamageReceiver	x	x	x	x	x	x	x
DamageDealer	x	x	x	x	x	x	x
FacePlayer	x	x	x	-	-	-	-
FaceJumpDirection	-	-	-	-	x	x	-
FollowNode	x	x	-	-	-	x	x
FollowNodeFlying	-	-	-	x	-	-	-
Health	x	x	x	x	x	x	x
Magic	-	-	x	-	-	-	-
HealthInfo	x	x	x	x	x	x	x
HumanoidBody	x	x	x	-	-	-	-
SlimeBody	-	-	-	-	x	x	-
MothBody	-	-	-	x	-	-	-
OgreBody	-	-	-	-	-	-	x
Jump	-	-	-	-	-	x	-
JumpPrecipice	-	x	-	-	-	-	-
JumpRandom	-	-	-	-	x	x	-
KnockBack	x	x	x	x	x	x	-
LootTable	x	x	x	x	x	x	-

Tabla 5.1: Tabla de componentes de enemigos. SK=Skeleton, GB=Goblin, GM=GoblinMage, MT=Moth, SM=Slime, SF=SlimeFast, OG=Ogre.

5.5.2. Solución final

Se va a crear una clase básica cuya única función es la capacidad de sentir la gravedad del planeta. La gravedad que sienten es modificable por instancia así que no se ha sacado fuera por ahorrarme el paso de añadir ese componente a todos. Cada enemigo que se cree heredará de esa clase y se le añadirán de uno en uno los componentes que necesite.

Esta solución ha arreglado el problema de que los enemigos dependieran de otros enemigos y ha proporcionado libertad absoluta para personalizar su comportamiento sin miedo a introducir fallos en otros enemigos.

No solo eso, gracias a la forma de añadir componentes del motor Godot, construir enemigos se hace sin escribir una sola línea de código (si el componente ya existe).

Como explicaré en el siguiente capítulo, esta fue la mejor decisión de todo el proyecto.

Capítulo 6

Desarrollo

Una vez realizado todo el análisis de requisitos y los diagramas iniciales que se pensaba que se iban a utilizar, me puse manos a la obra con el proyecto.

6.1. Inicio del proyecto: Arte del juego

Una de las decisiones más difíciles a tomar cuando se realiza un videojuego, es elegir el tipo de arte que se va a usar. Ya que en este proyecto lo que quiero mostrar son mis habilidades para resolver problemas de ingeniería de software más que las artísticas que pudiera tener, he decidido usar diseños minimalistas usando el arte pixelado. La mayoría de los dibujos los he realizado dentro de cuadros de 32x32 o 16x16 píxeles.

Para dibujar decidí usar el programa Aseprite, la mejor herramienta de código abierto que he probado hasta ahora para realizar este tipo de arte. Además, tiene la habilidad de separar las capas del dibujo en archivos individuales, lo que permitirá animar más adelante cada parte de los personajes.

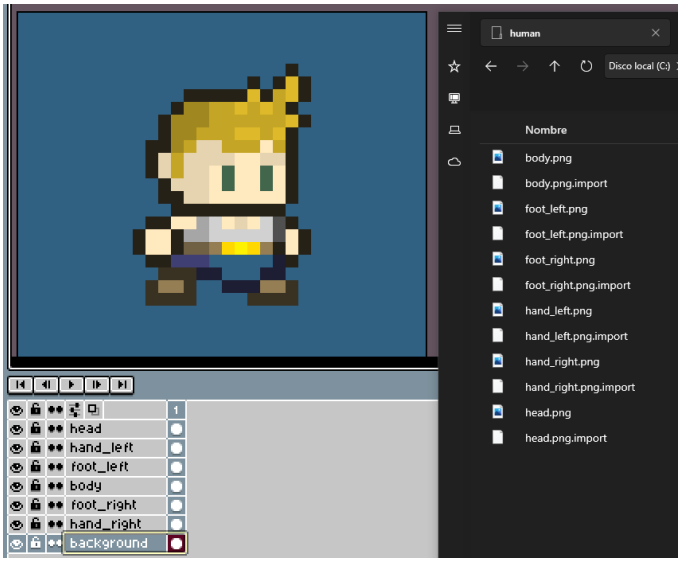

Figura 6.1: Dibujo del personaje jugable.

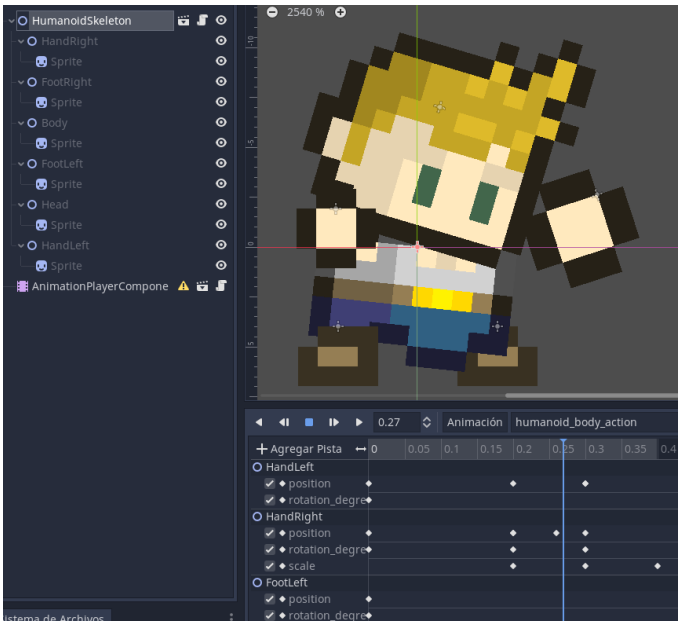

Figura 6.2: Ejemplo de una animación utilizando las capas que se han exportado de la imagen.

```
# Exportar las capas en imágenes individuales
aseprite.exe -b archivo.ase --save-as carpeta/{layer}.png
```

Lo mejor de haber dividido la imagen por estas capas, es que para los enemigos humanoides puedo reutilizar las animaciones que he hecho. De esta forma, solo hay que sustituir las imágenes por las del personaje que corresponda y reutilizar las animaciones que se le quieran poner. Para los enemigos que no tienen una estructura humanoide hay que crear animaciones propias para los tipos de monstruos que sean.

Figura 6.3: Dibujos de los enemigos ordenados por niveles en los que aparecen. Nivel 1 (skeleton, moth), nivel 2 (slime, slime fast), nivel 3 (goblin, goblin mage), nivel 4 (ogre).

6.1.1. Entorno

Para dibujar los entornos utilicé los *Tileset*, una herramienta que permite crear niveles usando los cuadraditos de una imagen. Dependiendo de como se pinten los cuadraditos por la pantalla y de qué otros cuadraditos tenga a su alrededor, el Tileset sabe que cuadrado poner específicamente para que las conexiones entre ellos queden bien.

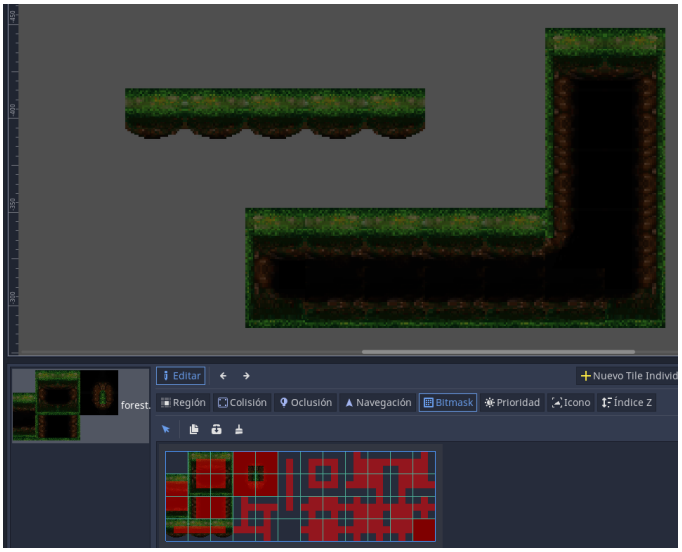

Figura 6.4: Ejemplo de uso de un Tileset.

Como se ha decidió usar plantillas para la creación de niveles aleatorios, usar el Tileset va a ser una buena forma de pintar los caracteres que se pongan en las plantillas.

6.2. PT1. Creación de niveles

Como ya expliqué en el capítulo 4 (Antecedentes), iba a utilizar plantillas con componentes randomizados para crear las estructuras del nivel. A continuación tenemos el ejemplo de una plantilla:

```
1 00000000000000000000000000000
2 00000000000000000000000000000
3 00000000000000000000000000000
4 00000000003333333300000000000
5 00000000002222222220000000000
6 00003333000000000000333300000
7 10002222000000000000222200000
8 10x0000000000000000000000d00
```

```
9  13333333333333333333333333333
10 11111111111111111111111111111
```

Hay 3 tipos de plantillas, las que se conectan horizontalmente y las que conectan con una subida o bajada. Cada plantilla representa una sala y lo que hace el algoritmo es recorrer cada carácter interpretando su significado:

- **x**: punto inicial donde aparecerá el personaje.
- **d**: salida del nivel.
- **0**: vacío, no se pintará nada.
- **1**: suelo.
- **2**: plataforma.
- **3**: componente al azar. El algoritmo colocará en esta casilla objetos interactuables, enemigos o lo dejará vacío. Dependiendo del nivel en el que nos encontremos habrá una probabilidad diferente de los elementos a mostrar.

Una vez que ha terminado de interpretar la plantilla, escoge otra al azar y se coloca en la posición adecuada para empezar a pintar la siguiente sala dependiendo si es horizontal o con cambio de elevación. Por cada nivel se interconectan 3 salas menos en el último nivel, que solo es una sala donde se lucha contra el jefe final.

6.2.1. Ejemplos de niveles

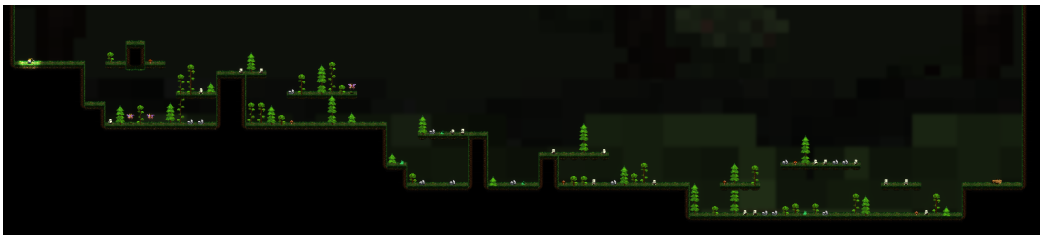

Figura 6.5: Nivel 1: Bosque verde.

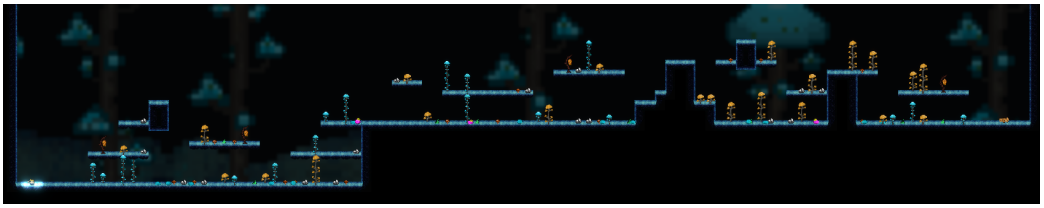

Figura 6.6: Nivel 2: Bosque de champiñones.

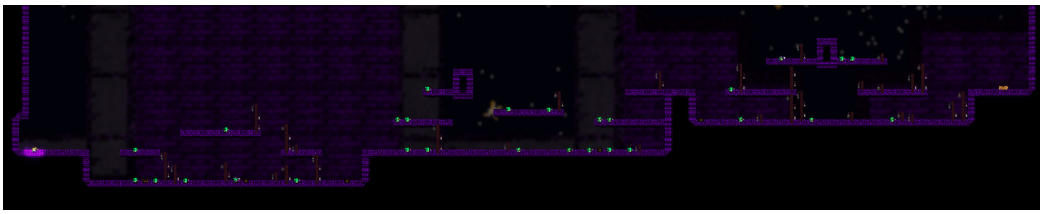

Figura 6.7: Nivel 3: Castillo.

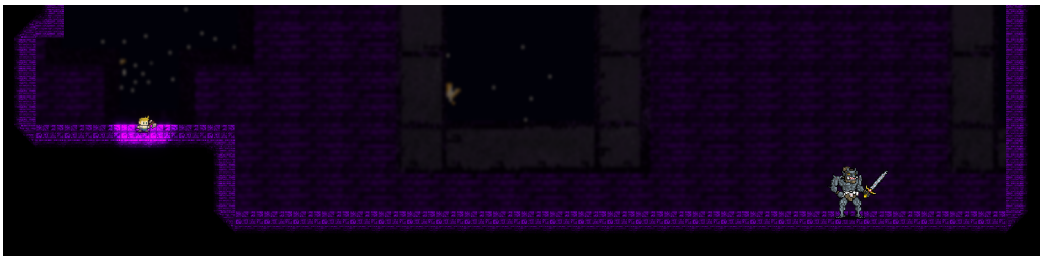

Figura 6.8: Nivel 4: Batalla final

6.2.2. Cómo elegir el componente randomizado

Cuando se lee en la plantilla el carácter *3*, hay que elegir dependiendo del nivel qué es lo que se tiene que poner ahí. Para ello, se le han dado diferentes pesos a los distintos elementos que pueden aparecer por nivel:

```
1 forest_spawner = {
2     null: 30,
```

```
3       skeleton: 10,
4       moth: 3,
5       herb: 4,
6       mushroom: 4,
7       mineral: 10,
8       tree: 35,
9   }
10
11  mushroom_spawner = {
12      null: 30,
13      slime: 7,
14      slime_fast: 1,
15      honey_comb: 2,
16      herb: 4,
17      mushroom: 10,
18      mineral: 10,
19      tree: 35,
20  }
21
22  castle_spawner = {
23      null: 30,
24      goblin: 10,
25      goblin_mage: 3,
26      tree: 10,
27      chest: 1,
28  }
29
30  final_level_spawner = {
31      ogre: 1,
32  }
```

Cuanto mayor sea el número del elemento, más probabilidad tendrá de que se escoja para aparecer en el mapa. Además, si el elemento es *null* no se creará nada en esa posición. Dejar huecos vacíos genera un nivel mucho menos recargado.

6.2.3. Problemas

A pesar de tener el máximo cuidado, durante el desarrollo de esta parte tuve la mala suerte de enfermarme de COVID-19. Estuve 2 semanas recuperándome

en las que no trabajé en el proyecto y un mes con una tos fuerte. Tuve la fortuna de que no me quedara ninguna secuela.

En cuanto a la implementación no tuve ningún problema en esta parte, por suerte todo salió según lo planeado en la fase de análisis.

6.3. PT2. Elementos interactuables

El jugador tiene que poder obtener recursos del entorno. Ahora que en la generación del mapa se tiene la opción de generar contenido aleatorio, se pueden añadir elementos que aparecerán al azar y que le darán recursos al personaje:

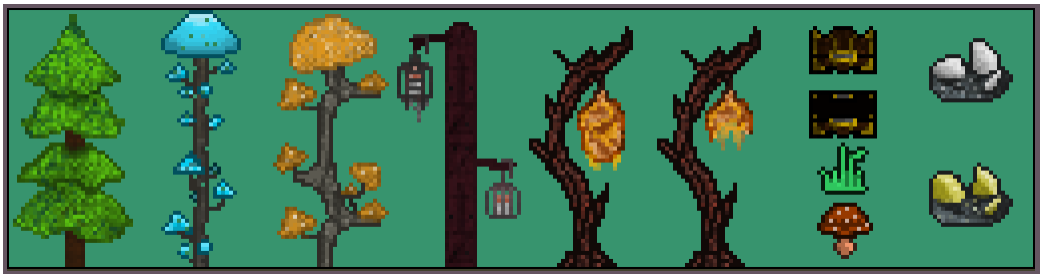

Figura 6.9: Objetos interactuables.

De izquierda a derecha: 4 tipos de árboles, panal de miel (normal y roto), cofre (abierto y cerrado), hierba, seta y minerales (hierro y oro). El funcionamiento es muy sencillo, el jugador realiza un ataque y el objeto detecta si ese ataque es del tipo correcto que este necesita para producir un efecto.

La idea es la siguiente:

- **Árbol:** para que se pueda talar se tiene que atacar con un hacha. Después de varios golpes caerá al suelo, desaparecerá y soltará madera.
- **Panal:** cualquier objeto, incluso las manos vacías, producirá que se rompa el panal y aparezca miel después de un par de golpes.
- **Cofre:** interactuar con lo que sea provocará que aparezcan objetos al azar.

- **Hierba:** golpearla hará que desaparezca y aparezca hierba que se puede recoger.
- **Seta:** lo mismo que la hierba pero con una seta.
- **Mineral de hierro:** golpearlo con un pico lo romperá y aparecerá hierro en el suelo que se puede recoger.
- **Mineral de oro:** necesita un pico más potente, el de hierro. Aparecerá oro que se podrá recoger.

En este punto todavía no se han implementado los objetos a recoger del suelo, simplemente los objetos interactuables están emitiendo una señal de los elementos que quieren que aparezcan. Es decir, romper un panal de miel envía la señal de que en esa posición tiene que aparecer miel. En el punto PT4. se implementará cómo se recoge esa señal y el objeto que tiene que aparecer.

6.4. PT3. Implementación de interfaces

Una de las razones por las que decidí utilizar Godot es por la facilidad de creación de interfaces. A continuación muestro como ejemplo el menú de pausa al que le añadí un control de volumen, ya que no se me había ocurrido cuando hice el análisis.

Figura 6.10: Menú de pausa.

No hubo un retraso significativo al desarrollar la parte de modificar el volumen porque en Godot en pocas líneas se puede implementar:

```
1 # Función que se activa al deslizar el slider del volumen
2 func _on_HSliderVolume_value_changed(value):
3     AudioServer.set_bus_volume_db(AudioServer.get_bus_index("Master"), value)
```

6.4.1. Problemas con la creación de objetos

Al principio quería desarrollar un menú que mostrase los objetos a la izquierda y los combinaciones que se podían hacer a la derecha.

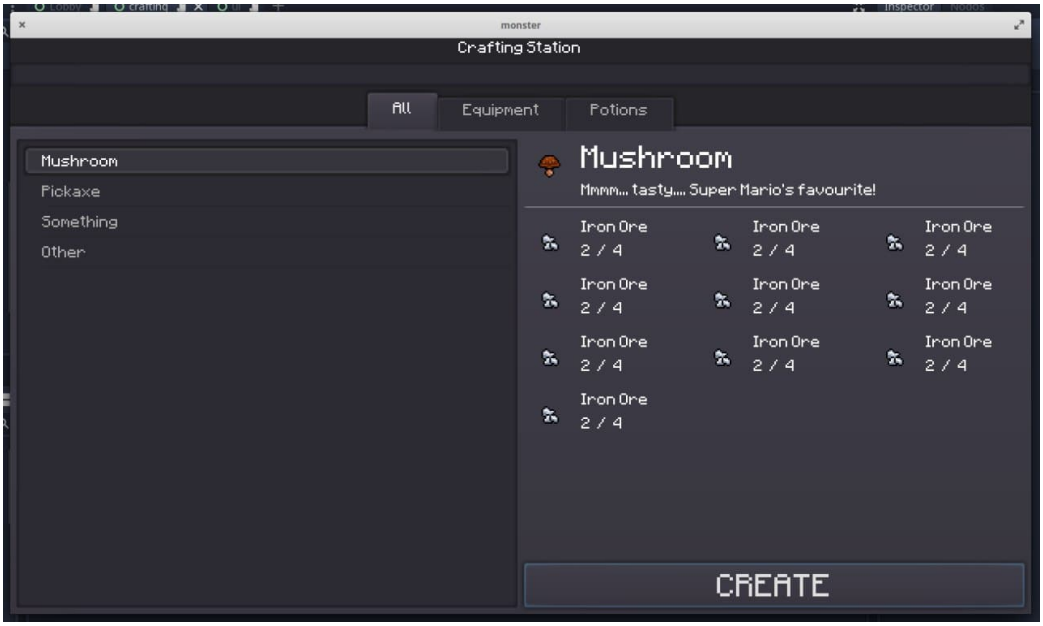

Figura 6.11: Antigua interfaz de creación de objetos.

Pronto deseché esta idea porque no me gustaba nada como se veía y además tapaba el nivel. Decidir empezar de nuevo a crear la interfaz fue una decisión difícil de tomar, pues retrasaría el avance y no garantizaría que me fuera a gustar más que esta primera solución.

Después de tomar como referencia otros videojuegos, decidí implementar una interfaz similar al videojuego Terraria. En la que la creación de objetos se muestra abajo a la izquierda.

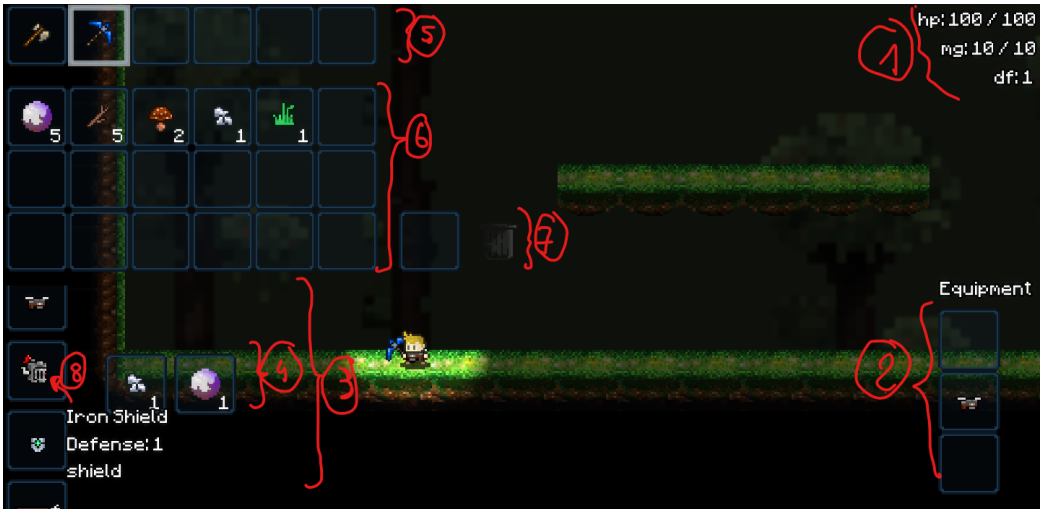

Figura 6.12: Interfaz del jugador durante el nivel cuando pulsa la tecla TAB.

- **1:** Muestra la información del personaje. hp (puntos de vida), mg (puntos de magia), df (defensa).
- **2:** equipamiento que puede llevar el personaje (casco, pechera o escudo).
- **3:** lista de objetos que se pueden crear con los objetos que se poseen en el inventario (6).
- **4:** lista de objetos que se consumirán del inventario al crear el objeto seleccionado en la lista (3).
- **5:** lista de objetos que se pueden poner en la mano. Moviendo la rueda del ratón se van seleccionando o pulsando un número del 1 al 6.
- **6:** lista de objetos del inventario. Se pueden mover entre la lista de objetos a tener en la mano o apilar los objetos del mismo tipo siempre que no sean herramientas o armadura.
- **7:** papelera para depositar objetos que no se necesitan.
- **8:** información que sale al pasar el cursor sobre un objeto.

La información de las recetas se guardan en un diccionario. En total hay 17 recetas, a continuación muestro el ejemplo de 2:

```
_recipes = {
    "gold_sword": {
        "gold": 1,
        "wood": 1
    },
    "potion_health": {
        "herb": 1,
        "honey": 1
    }
}
```

Para obtener la lista de las recetas que se pueden hacer se pasa la lista de los objetos del inventario del jugador al RecipeManager, el gestor de recetas. Este va mirando receta a receta las que se puedan hacer y devuelve una lista con las que son posibles. Esa lista es la que se muestra en el punto 3 de la imagen anterior.

6.5. PT4. Objetos utilizables

En el PT2, elementos interactuables, se habían implementado los recursos con los que el jugador podía interactuar pero no aparecían los objetos que el jugador podía recoger en el mapa. Esta parte se ha decidido hacer después de implementar las interfaces para poder equipar estos objetos en el inventario y facilitar las pruebas. Es más sencillo probar los objetos si ya se dispone de un inventario con varias casillas donde ir almacenándolos.

Hacer que los objetos aparezcan en el mapa es trivial. Lo hace la clase *Stage* cuando recibe la señal de que hay que hacer que aparezca un objeto. Recibe el nombre del objeto y crea el objeto pasándole el identificador. Esto hace que el propio objeto pregunte por la imagen que necesita mostrar al gestor de objetos.

Cuando el jugador se acerca al objeto se añade a su inventario. Estos son los objetos que existen:

Figura 6.13: Objetos que pueden usarse en el juego.

- **Herramientas:** pico (A6) y pico de hierro (B6). Sirven para minar minerales.
- **Objetos combinables:** seta (A1), miel (B1), hierba (C1), madera (D1), gelatina (A5). También están los núcleos (B5, C5), que sirven para combinarlos con minerales para crear armaduras.
- **Armadura de hierro:** (B2, C2, D2) cada parte incrementa un punto la defensa.
- **Armadura de oro:** (B3, C3, D3) cada parte incrementa dos puntos la defensa.
- **Espadas:** hierro (C4) +4 de ataque, oro (B4) +6 de ataque.
- **Magia:** (C6) lanza una bola de energía si se tiene puntos de magia suficientes.
- **Lanzamientos:** tenemos un arco (D5) y una flecha (D6), pero también una lanza (A7).
- **Pociones:** poción de salud (A4) recupera 5 puntos de vida, poción de magia (B4) recupera 5 puntos de magia.

La implementación de los efectos de los objetos se ha realizado mediante las animaciones. Lo primero que se hace es mirar el objeto que se tiene equipado, por ejemplo el bastón de magia. Por lo tanto, al pulsar el click derecho del ratón para activar su efecto, se selecciona esa animación y se ejecuta la función de activar el efecto, que en este caso es lanzar una bola de magia.

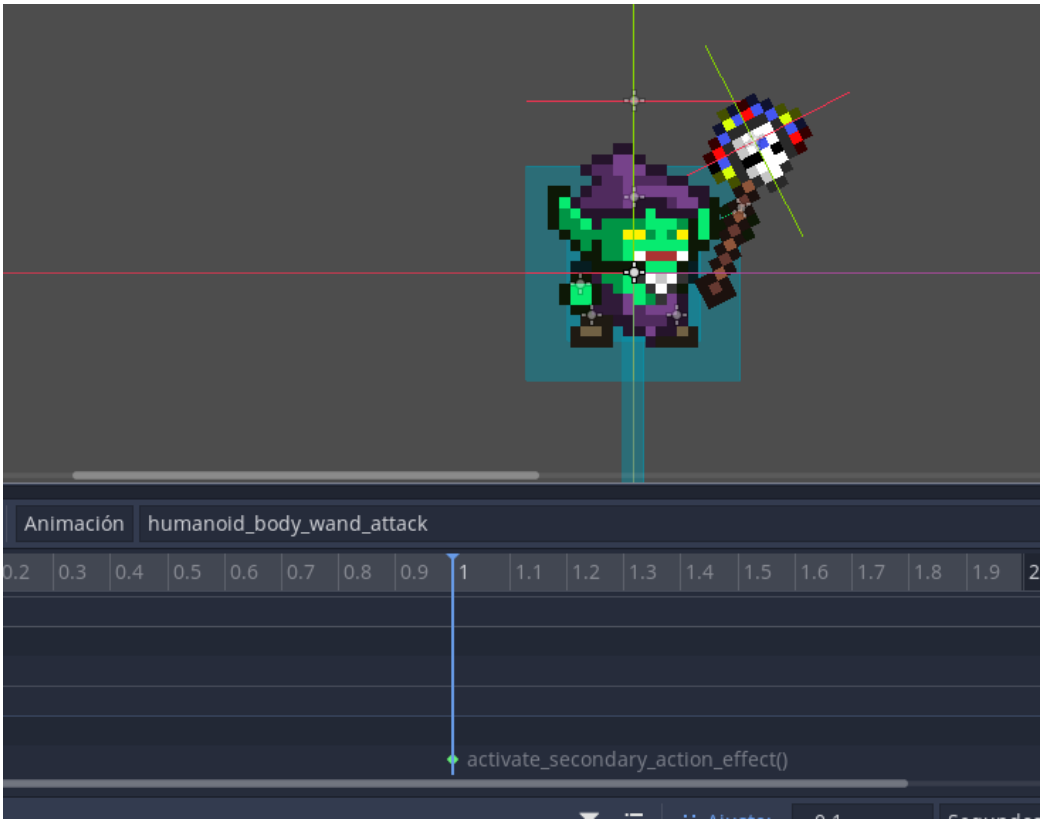

Figura 6.14: Ejemplo de activación de un objeto.

Como se puede comprobar, hay una pista en la animación que invoca la función *activate_secondary_action_effect()*. Si se tiene equipado un bastón de magia intentará lanzar magia, si tiene una poción intentará recuperarse vida, si tiene un arco intentará lanzar una flecha...etc. Es un sistema muy sencillo que se podría ampliar en el futuro con facilidad.

6.6. PT5. Creación de enemigos

Esta parte ha sido la más divertida de implementar. Como he explicado en el capítulo del análisis y diseño, decidí crear una entidad básica a la que solo le afectaba la gravedad. La idea principal era crear enemigos añadiéndole comportamientos o componentes a esa entidad.

Hacerlo de esta manera en lugar de un árbol extenso de herencia ha permitido una gran flexibilidad para crear enemigos únicos, pero que son capaces de compartir comportamientos con otras entidades.

6.6.1. Comportamiento de los enemigos

Figura 6.15: Dibujos de los enemigos ordenados por niveles en los que aparecen. Nivel 1 (skeleton, moth), nivel 2 (slime, slime fast), nivel 3 (goblin, goblin mage), nivel 4 (ogre).

- **Skeleton**: cuando detecta al jugador se va acercando hacia él para intentar tocarle y así disminuir su vida.
- **Moth**: mismo comportamiento que el esqueleto pero se acerca volando. Por lo tanto, aunque el jugador salte para escapar la polilla podrá seguirle.
- **Slime**: salta aleatoriamente hacia los lados y resbala mucho en el suelo.
- **SlimeFast**: igual que el Slime pero si detecta al jugador saltará hacia él.

- **Goblin**: si detecta al jugador lo perseguirá y si está muy cerca le atacará con su cuchillo. Además, cuando detecta que se va a caer de una plataforma salta primero para realizar una caída espectacular.
- **GoblinMage**: cuando detecta al jugador le atacará desde lejos lanzándole magia.
- **Ogre**: cuando detecta al jugador se abalanza periódicamente hacia él, quitándole gran cantidad de vida. Además, mueve la cabeza haciendo como que está mirando al jugador.

Para todos los enemigos que son capaces de detectar al jugador, si el jugador se aleja lo suficiente dejarán de seguirle e intentar atacarle.

6.6.2. Implementación

Los componentes se escuchan entre sí para realizar comportamientos determinados. Además, cada componente es parametrizable para ajustar las preferencias que queramos que tengan las entidades (velocidad, rango de detección... etc). A continuación se va a mostrar un ejemplo del uso de componentes para construir un Goblin, ya que es la entidad más compleja del juego:

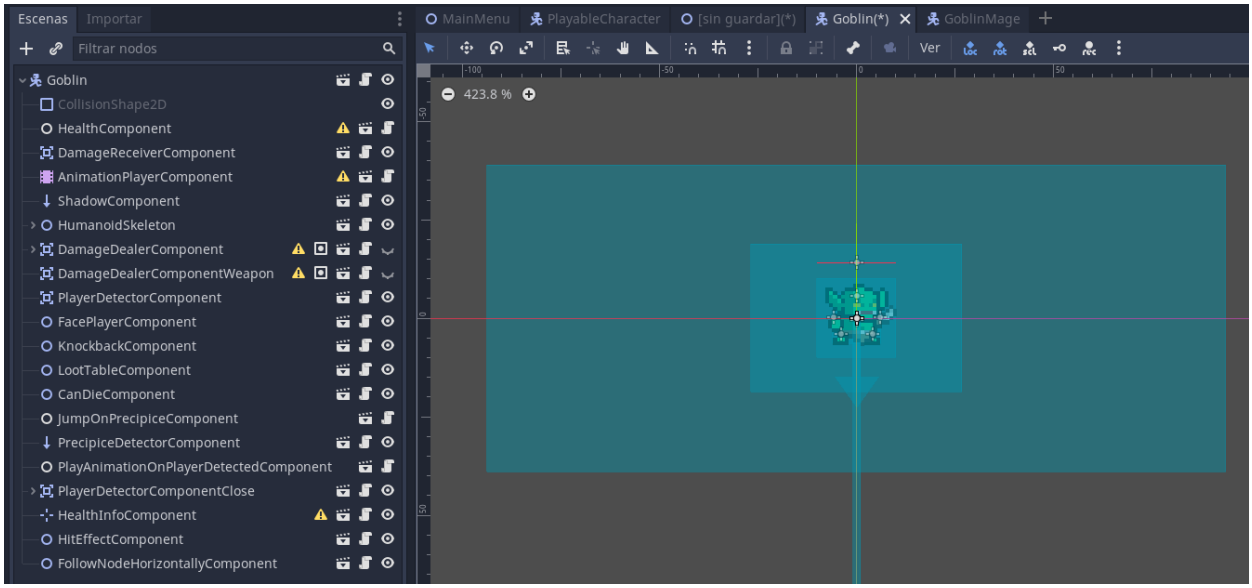

Figura 6.16: Ejemplo de los componentes que tiene un goblin.

- **CollisionShape2D:** sirve para que detecte el suelo y las paredes.
- **HealthComponent:** lleva la cuenta de la vida. Está suscrito a *DamageReceiverComponent* para saber cuando le están haciendo daño y gestiona la vida que le quitan. Además, manda una señal cuando la vida cambia.
- **DamageReceiverComponent:** detecta las colisiones con las cosas que puede detectar como por ejemplo las armas del personaje. Envía una señal cuando detecta que algo ha entrado a él.
- **AnimationPlayerComponent:** se encarga de reproducir las animaciones. Escucha al nodo de la entidad principal y dependiendo de su vector de movimiento decide que animación usar:
  - Si el vector es (0, 0) ejecuta la animación de descansar.
  - Si el vector tiene el componente y como 0 pero el x es distinto a 0 ejecuta la habilidad de andar.
  - Si el vector tiene un componente y distinto a 0 interpreta que está en el aire y ejecuta la animación de salto.
- **ShadowComponent:** detecta donde está el suelo para poner una sombra.
- **HumanoidSkeleton:** cuerpo del personaje.
- **DamageDealerComponent:** un área capaz de ser detectada por otras áreas. En este caso corresponde al cuerpo, es decir, que si el jugador toca esta área sufrirá daño.
- **DamageDealerComponentWeapon:** área que corresponde al cuchillo, si el jugador detecta esta área le quitará vida.
- **PlayerDetectorComponent:** corresponde al área azul más grande de la imagen. Se encarga de detectar al jugador y envía una señal cuando lo hace con su información.
- **FacePlayerComponent:** está escuchando a *PlayerDetectorComponent* para cambiar la dirección a la que mira al personaje.
- **KnockbackComponent:** escucha a *DamageReceiverComponent* y provoca un retroceso cuando recibe la señal de que ha recibido daño.
- **LootTableComponent:** guarda la información de los objetos que tiene que soltar al morir.

- **CanDieComponent:** escucha a *HealthComponent* y cuando recibe la señal de que ha muerto se encarga de hacer que aparezcan los objetos en el mapa y de que desaparezca el personaje.
- **JumpOnPrecipiceComponent:** escucha a *PrecipiceDetectorComponent* para hacerle saltar en el momento preciso.
- **PrecipiceDetectorComponent:** manda una señal cuando detecta que va a llegar a un precipicio.
- **PlayAnimationOnPlayerDetectedComponent:** escucha a un *AnimationPlayerComponent*, en este caso a *PlayerDetectorComponentClose*, para ejecutar una animación cuando detecte al personaje. Esto permite que el goblin pueda atacar con el cuchillo cuando detecte al personaje cerca.
- **PlayerDetectorComponentClose:** detecta al personaje. Es la cajita azul mediana de la imagen.
- **HealthInfoComponent:** escucha a *HealthComponent*, cuando recibe la señal de que la vida ha cambiado enseña un número de la cantidad de vida que ha perdido.
- **HitEffectComponent:** escucha a *DamageReceiverComponent*, pinta el cuerpo de color blanco por un momento, sirve como feedback para que el jugador sepa que ha provocado daño al enemigo.
- **FollowNodeHorizontallyComponent:** escucha a *PlayerDetectorComponent*, cuando recibe la señal de que se ha detectado al jugador hace que se mueva hacia la dirección de lo que ha detectado.

Los demás enemigos poseen variaciones más sencillas que el goblin. Las babosas por ejemplo tienen un componente de salto, la polilla tiene un componente para moverse por el aire (y su gravedad se ha ajustado a cero), el ogro tiene un componente para abalanzarse sobre el jugador y el jugador tiene un componente para detectar las teclas del teclado para poder moverlo.

Lo más interesante de estos componentes es que se desarrollan por separado. Al construir los enemigos solo hay que añadirlos a la entidad base e indicar a qué otros componentes tienen que escuchar. He exportado cada variable que necesitan al editor y esto permite que no haya que escribir código para crear enemigos. Con arrastrar el componente a la variable es suficiente.

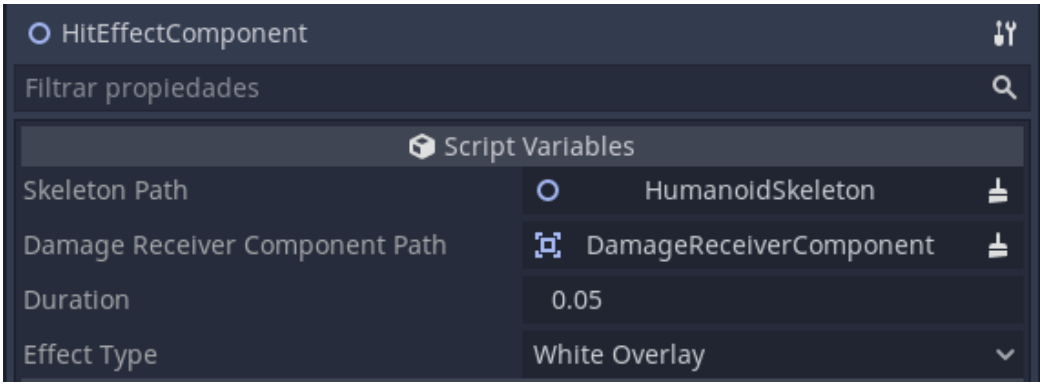

Figura 6.17: Ejemplo de las variables que pide el componente *HitEffectComponent* para funcionar.

6.6.3. Ejemplo del funcionamiento de un componente

Para ilustrar la flexibilidad de los componentes voy a mostrar como funciona *KnockbackComponent*, el componente que se encarga de aplicar un retroceso a los enemigos cuando reciben daño.

El primer paso es cumplir los requisitos del componente. Al añadirlo a un enemigo se le deben asignar en sus variables los elementos que pide. En este caso necesita un *KinematicBody2D* y un *DamageReceiverComponent*. El primero es el componente al que se le va a aplicar el retroceso y el segundo es el componente al que se va a suscribir para detectar cuándo ha recibido daño. De esta manera, *KnockbackComponent* aplicará retroceso cada vez que detecte la señal de que se ha recibido daño.

Como se puede observar, los componentes realizan funciones muy específicas y tienen un conocimiento limitado de todo el sistema. Con muy poquito código se puede crear uno y todo esto permite que cada parte esté aislada del resto. Además, si hay fallos se pueden reconocer rápido de dónde provienen. Por ejemplo, ¿el goblin no salta cuando llega al final del precipicio? Entonces el problema estará o en el componente que detecta los precipicios o en el que ejecuta el salto.

Además, a todos los componentes les he añadido la función especial de Godot *_get_configuration_warning()*, que la he usado para mostrar un *warning* cuando no se hayan conectado correctamente los componentes que requiere. Reduciendo de esta forma la posibilidad de que se introduzcan errores por una mala inicialización.

Capítulo 7

Verificación y evaluación

Al realizar cada prototipo, se necesita que pase por una fase de pruebas antes de continuar con el desarrollo. Esto es necesario para evitar la acumulación de errores en fases futuras del proyecto.

Una de las herramientas típicas para la creación de pruebas en proyectos de software son los tests de unidad. Mediante este tipo de tests se comprueba que partes específicas del programa funcionan correctamente. Lo mejor de este tipo de pruebas es que son automáticas, se pueden ejecutar periódicamente o incluso debido a un evento como podría ser el guardado del documento.

Por desgracia, Godot no posee una herramienta de tests de unidad oficial. Pero pensándolo mejor, debido a la naturaleza de este proyecto lo que tienen más sentido son los tests de integración, en los que se comprueban que diferentes sistemas interaccionan bien entre sí, como podrían ser los componentes de los enemigos.

Para los tests de integración lo que se ha hecho es iniciar el juego y comprobar visualmente que todo funciona correctamente. Al fin y al cabo, si lo que se quiere probar es que el juego funciona bien lo mejor es comprobarlo jugando.

A continuación se muestra una serie de tablas con los tests realizados en cada prototipo. El color verde significa que ha ido correctamente, rojo es fallido y amarillo no del todo correcto. Al final del capítulo se detalla este último resultado de los tests.

Tests PT1. Creación de niveles		
Descripción	**Result**	**Solved**
1. El suelo se genera correctamente.		
2. Las plataformas se generan correctamente.		
3. Las salas se conectan bien entre sí.		
4. El jugador aparece en el mapa.		
5. La salida aparece en su lugar en el mapa.		
6. El jugador puede ir a otro nivel.		
7. Los niveles son aleatorios.		

Tabla 7.1: Pruebas asociadas a la generación del nivel.

Tests PT2. Elementos interactuables		
Descripción	**Result**	**Solved**
1. Los elementos aparecen en el mapa.		
2. Se puede talar árboles.		
3. Solo talar con hachas.		
4. Se puede minar minerales.		
5. Minar hierro con picos.		
6. Minar oro solo con pico de hierro.		
7. Recoger setas.		
8. Recoger hierbas.		
8. Recoger miel.		

Tabla 7.2: Pruebas asociadas a los elementos interactuables.

Tests PT3. Creación interfaces		
Descripción	**Result**	**Solved**
1. Menú de inicio: iniciar juego.		
2. Menú de inicio: salir del juego.		
3. Menú de inicio: modificar volumen.		
4. Menú de pausa: modificar volumen.		
5. Menú de pausa: mostrar información de controles.		
6. Inventario: equipar armadura.		
7. Inventario: seleccionar objeto.		
8. Inventario: mover objeto.		
9. Inventario: apilar objeto.		
10. Inventario: eliminar objeto.		
11. Inventario: ver información del objeto.		
12. Inventario: crear objeto.		

Tabla 7.3: Pruebas asociadas a la implementación de las interfaces.

Tests PT4. Objetos utilizables		
Descripción	**Result**	**Solved**
1. Los objetos interactuables sueltan objetos utilizables.		
2. Se pueden tomar pociones.		
3. Se puede atacar desde lejos.		
4. Se puede atacar con cualquier arma.		
5. Se puede atacar sin arma.		
6. Cada arma hace su daño correspondiente.		

Tabla 7.4: Pruebas asociadas a la utilización de objetos.

Tests PT5. Creación de enemigos		
Descripción	**Result**	**Solved**
1. Pierden vida al ser atacados.		
2. Muestran vida al ser atacados.		
3. Detectan al jugador.		
4. Dejan de detectar al jugador si se aleja.		
5. Se mueven hacia el jugador.		
6. Pueden morir.		
7. Sueltan objetos al morir.		
8. El jugador puede morir.		
9. Vencer al enemigo final termina la partida.		
10. La sombra se ve bien.		
11. Los proyectiles los lanzan correctamente.		
12. Saltan correctamente.		
13. Cuando reciben daño parpadean en blanco.		
14. Cuando reciben daño retroceden.		

Tabla 7.5: Pruebas asociadas a la creación de enemigos.

7.1. Elementos notables

Hay algunos tests que no se han definido como errores pero que no están totalmente correctos:

- **PT2.1 Los elementos aparecen en el mapa:** se ha detectado que cuando se tala un árbol y este cae contra la pared, el jugador no puede acceder a toda la madera. Como sobra la madera en todos los niveles no lo he considerado un error.
- **PT3.10 Eliminar objeto:** si se intenta eliminar un objeto no apilable del mismo tipo cuando ya está la papelera llena con ese objeto, no se puede eliminar. Pero sí se puede si se pone un objeto de otro tipo y después el que se quiere eliminar. No es muy molesto así que no le he considerado error.

Capítulo 8

Conclusiones y trabajo futuro

Este trabajo de fin de grado ha sido el proyecto más grande que he hecho nunca. Ha puesto al límite todos los conocimientos que he adquirido durante la carrera y ha hecho que vaya un paso más allá para que diera lo mejor de mí.

Estoy muy satisfecho con el trabajo realizado y me he quedado con ganas de seguir trabajando en ello. No cabe duda de que continuaré puliendo este proyecto.

8.1. Objetivos

El objetivo principal de este proyecto era crear un videojuego en el que cada partida fuera diferente de la anterior. Para ello, los niveles se tenían que generar proceduralmente y es lo que se ha hecho. Cada partida genera niveles diferentes y coloca los recursos/enemigos aleatoriamente, así que debido a la gran variedad de posibilidades, no se van a jugar dos partidas iguales.

Otro objetivo era adquirir nuevos conocimientos en el mundo del desarrollo de los videojuegos. La investigación que se ha tenido que llevar a cabo para cumplir los requisitos del proyecto han hecho que haya tenido que aprender muchísimo, sobre todo de buenas prácticas de diseño y álgebra vectorial.

Y como objetivo adicional también quería demostrar que se puede crear un juego utilizando únicamente software de código abierto. He utilizado Godot, Aseprite y un Portátil con Linux así que como podemos ver en las imágenes a continuación, es posible.

8.1.1. Imágenes del juego terminado

Figura 8.1: Nivel 1: Bosque verde con el inventario abierto.

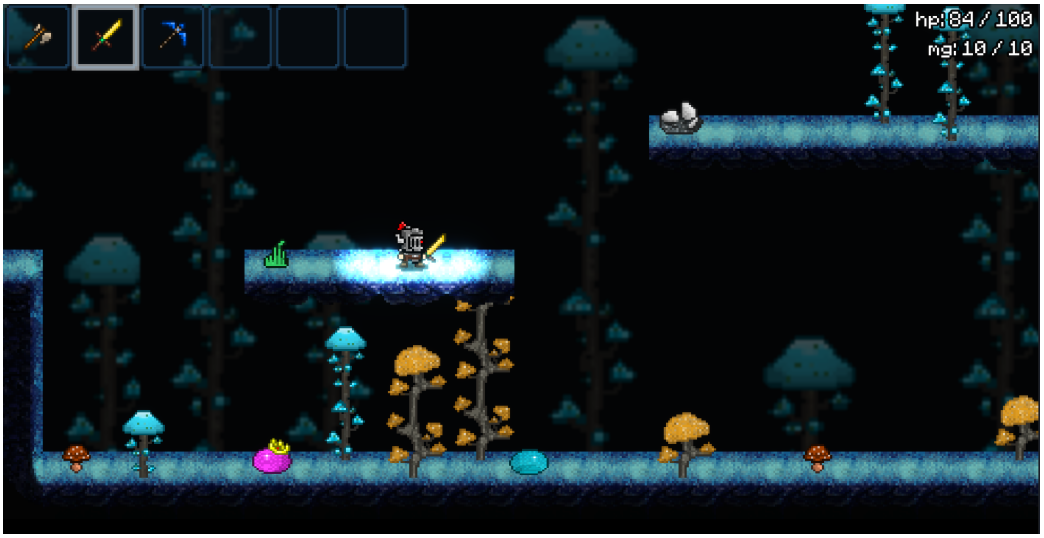

Figura 8.2: Nivel 2: Bosque de champiñones con el inventario cerrado.

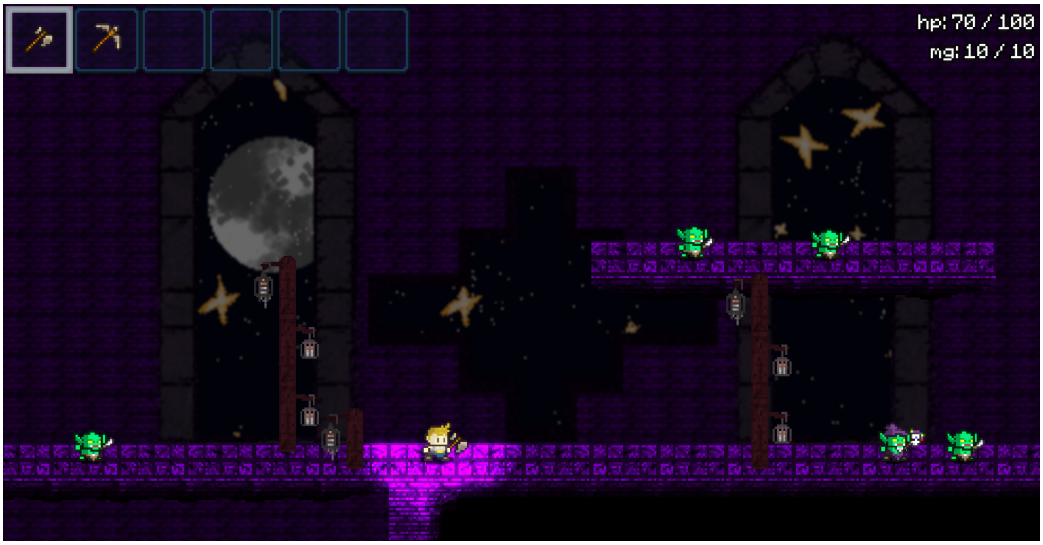

Figura 8.3: Nivel 3: Castillo.

Figura 8.4: Nivel 4: Batalla final.

8.2. Gestión de riesgos

A continuación, se va a evaluar el éxito de los planes de prevención y contingencia definidos.

1. **RH1. Avería del equipo de trabajo**: tuve mucho cuidado de no acercar comida o bebida cerca del portátil y nunca lo puse en lugares inestables desde donde pudiera caerse. Además, con cada cambio que hice en el proyecto resguardé copias de seguridad en el repositorio de código. A día de hoy, no concibo trabajar sin un programa de control de versiones donde resguardar el código así que en el futuro seguiré utilizándolo para futuros proyectos.

2. **RH2. Robo del equipo de trabajo**: nunca tuve la necesidad de sacar el equipo de casa y siempre cerré con llave la puerta de casa antes de dormir, así que las probabilidades de que el portátil fuera robado disminuyeron mucho.

3. **RP3. Poco tiempo disponible**: un riesgo que se terminó cumpliendo, ya que ir a la universidad, trabajar, ir a la academia y después estudiar en casa me quitaba la mayor parte del tiempo disponible del día. El plan de prevención consistía en gestionar el tiempo en el calendario, respetarlo y planificar descansos. Sin embargo, en época de exámenes, las horas de estudio eclipsaron el tiempo disponible para el desarrollo de este proyecto. Al final tuve que recurrir al plan de contingencia que consistía en sacrificar el tiempo libre, principalmente fines de semana, para continuar el progreso. El problema fue que muchos fines de semana también tenía que estudiar y después de trabajar en el proyecto no tenía prácticamente tiempo libre, lo que produjo que acabase muy agobiado. Es cierto que el plan de contingencia funcionó, pero si tuviera que hacer este proyecto de nuevo, si tuviera tan poco tiempo libre hubiera rebajado el alcance para tener menos tareas que hacer o hubiera definido menos horas por semana aunque se retrasase mucho la fecha de finalización.

4. **RP1. Contagiarse de COVID-19**: terminé contagiándome a pesar de tomar todas las medidas de prevención recomendadas. Probablemente por las aglomeraciones que se producen en el transporte público, donde mantener la distancia de seguridad es complicado y hay poca ventilación. El plan de contingencia consistía en cumplir las indicaciones sanitarias para recuperarme lo antes posible y lo hice, con la suerte de que mi enfermedad fue leve. Si volviera a verme en esta situación intentaría no usar el transporte público.

5. **RD1. Perfeccionismo innecesario**: el plan consistía en no dedicar más horas de las estimadas si ya se tenía un producto mínimo viable. Si bien es cierto que se me ocurrieron varias mejoras para varios sistemas, al ser ya funcionales no les dediqué más tiempo del necesario y di las tareas por

concluidas. En el futuro volveré a utilizar esta forma de trabajo, ya que ha permitido que pudiera avanzar con un producto funcional en lugar de retrasar el avance por la reimplementación de sistemas que aunque no sean perfectos, cumplen con su propósito.

6. **RD2. Errores en la planificación**: tuve errores en la planificación sobre todo en el diseño de las interfaces, porque no me gustaba como quedó el menú de combinación de objetos, y en el desarrollo de los enemigos, porque para cumplir los requisitos de los enemigos tuve que crear más componentes de los que había imaginado. Para que no vuelva a pasar, debería haber pasado más tiempo pensando en todos los tipos de componentes que iban a hacer falta. Por ejemplo, no había pensado en los componentes que mostraban los puntos de vida que se restan a los enemigos al atacarlos, ni el componente que cambia el color del enemigo al recibir ataques para que el jugador sepa que les está haciendo daño.

7. **RD3. Bugs en el motor de desarrollo**: Siempre investigué primero si la solución que había pensado podía implementarse en el motor antes de ponerme con la tarea y por suerte no hubo ningún bug en Godot que me impidiera continuar.

8. **RP2. Otras enfermedades**: a parte de COVID-19, no me enfermé de nada notorio. Imagino que al usar la mascarilla y cumplir las medidas de prevención evité enfermarme de enfermedades estacionales.

8.3. Revisión de la estimación inicial

Al final se ha tardado 26,5 horas más en terminar el proyecto, ascendiendo a un total de 304,5 horas frente a las 278 que se habían estimado.

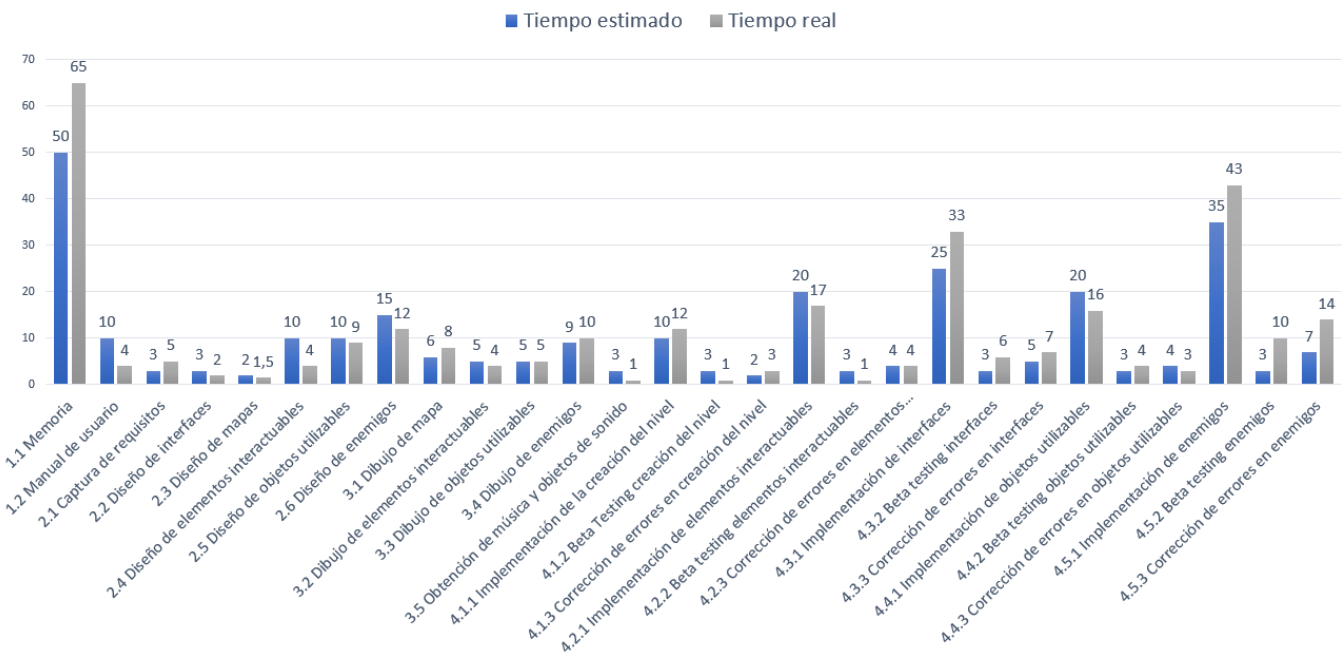

Figura 8.5: Tiempo estimado VS tiempo real.

Gastos asociados al proyecto en €	
Descripción	**Total**
ASUS GL552VW	25,8€
Aseprite	16,79€
Salario (22€/hora)	304,5 * 22 = 6699 €
TOTAL	**6741,59**

Tabla 8.1: Gastos asociados al proyecto.

La diferencia de gastos real contra la estimada: 6741,59 - 6152,86= **588,73**. Hay un sobrecoste del 9,5 %, lo que significa que habría que vender 140 copias más de las esperadas para cubrir los gastos del proyecto. Es decir, antes de empezar a generar beneficios tendrían que venderse 1629 copias.

8.3.1. Fecha de finalización

Debido a las semanas de retraso por haber padecido COVID-19, el poco tiempo sobrante que tenía después de trabajar y estudiar para los exámenes, la fecha final se ha retrasado mucho. La idea principal era trabajar 2 horas cada día laboral de la semana pero al final se ha trabajado sobre todo durante el fin de semana por la falta de tiempo. Haciendo que la fecha de finalización del desarrollo alcance el 29 de junio.

8.4. Trabajo futuro

Durante el desarrollo me he encontrado con varios errores de ejecución por errores de tipado. GDScript, el lenguaje de desarrollo de Godot, usa tipado dinámico y no avisa si se asignan valores con el tipo incorrecto hasta que se da el error en tiempo de ejecución. Es cierto que en futuras versiones están implementando tipado opcional pero no es tan seguro como podría ser el tipado estático. Por esa razón me plantearía usar C# para futuros proyectos con este motor de videojuegos.

Otra cosa importante sería la utilización de bases de datos para almacenar la información. Si bien es cierto que con la poca cantidad de objetos que hay no merecía la pena, si se sigue trabajando en el proyecto esto ayudaría a escalar más fácilmente. Más de una vez me he confundido poniendo el id de los objetos en los JSON relacionados y han fallado cosas porque no encontraba el id.

Una mejora interesante sería utilizar el nodo de Godot *VisibilityNotifier2D*. Este nodo permite que no se procesen los objetos si no están dentro de la cámara. Sería una buena optimización y además haría que al llegar a las úlitmas salas del nivel 2, las babosas no estuvieran todas ya en el suelo, por los saltos que han ido dando hasta que el jugador ha llegado.

También ha sido muy constructivo pedir feedback a la gente que quería probar este juego. Me han dado ideas muy buenas con las que mejorar, como:

- La magia es demasiado poderosa porque de un solo ataque puedes acabar con todos los enemigos del suelo, ya que la magia traspasa enemigos y paredes.
- Habría que aumentar la fricción contra el suelo del personaje para que

desacelere más rápido al moverse. Puede dar una sensación como andar sobre el hielo tal como está.

- Si te mueves durante las animaciones de ataque los pies no se mueven así que parece que vuelas.
- Al crear cosas como flechas estaría bien poder crear varias a la vez en lugar de tener que ir de una en una.
- Añadir personalización al personaje, para cambiarle el género, color de piel, pelo o ropa.

En resumen, el esfuerzo invertido ha merecido la pena con el producto final que se ha obtenido. Me produce mucha alegría haber acabado mi etapa universitaria con un proyecto así.

Anexo A

Manuales

Condición para ganar: avanzar por los niveles hasta llegar al nivel final y derrotar al jefe final.

Condición para perder: los puntos de vida del personaje llegan a cero. Si esto ocurre la partida se termina y se de opción de empezar de nuevo o salir.

A.1. Controles

Se utiliza el teclado y el ratón.

- A D: mover al personaje izquierda o derecha.
- W: pasar a la siguiente pantalla cuando se ha llegado al final del nivel.
- Espacio: saltar, dos pulsaciones para doble salto.
- Tab: abrir/cerrar inventario.
- **Números del 1 al 6 o rueda del ratón**: sujetar un objeto concreto entre los disponibles en la barra de acción.
- **Mover el ratón**: mueve la cámara.
- **Click izquierdo del ratón**: ataque principal.

- **Click derecho del ratón**: ataque secundario que varia dependiendo del arma.

A.2. Ordenar inventario de objetos

Después de abrir el inventario con Tab:

- **Reordenar objetos**: clic sobre el objeto para cogerlo y click sobre otra casilla para dejarlo o intercambiarlo por el objeto que halla.
- **Apilar objetos**: dejar un objeto sobre otro del mismo tipo.
- **Dividir objetos**: con un objeto agarrado pulsar click derecho sobre la casilla deseada para dejar el objeto de uno en uno.
- **Eliminar objeto**: dejar un objeto sobre la casilla con el icono de cubo de la basura.

A.3. Combinaciones

Las combinaciones de objetos permiten obtener objetos nuevos. Realizar combinaciones es vital para obtener equipo más defensivo, realizar más daño a los enemigos u obtener pociones curativas.

A.3.1. Cómo combinar objetos

Hay que pulsar **TAB** para abrir el inventario. Si se poseen objetos capaces de combinarse aparecerá una lista con los resultados de las combinaciones posibles debajo del inventario. Al clickar sobre una combinación se restarán automáticamente los objetos necesarios para crearlo y tendremos el resultado agarrado, que podremos posicionarlo en el hueco deseado del inventario.

Bibliografía

[1] Gamerdic. *Definición de motor de juego*. URL: http://www.gamerdic.es/termino/motor-de-juego. (accedido: 25/11/2020).

[2] GBJAM. *Especificaciones del GBJAM*. URL: https://itch.io/jam/gbjam-7. (accedido: 25/11/2020).

[3] Darius Kazemi. *Spelunky Generator Lessons Part 2: Generating the rooms*. URL: http://tinysubversions.com/spelunkyGen2/. (accedido: 04/11/2020).

[4] Unity Technologies. *Obtén mucho más con Unity*. URL: https://unity.com/es/solutions/game. (accedido: 04/11/2021).